DevAssistant Documentation

Release 0.9.2

Bohuslav Kabrda, Petr Hracek

October 04, 2014

DevAssistant - making life easier for developers

Contents

1.1 User Documentation

1.1.1 Subtopics

DevAssistant and Docker.io

Note: this document is under construction. The described features are not yet implemented in DevAssistant and might change significantly before version 0.9.0 is released.

Docker is "an open source project to pack, ship and run any application as a lightweight container".

A container is basically a lightweight virtual machine, that has all the dependency installation and system setup done inside it, so they don't affect your system.

This page summarizes Docker usage workflow during project development/deployment, as well as instructions on how to make the steps painless through DevAssistant.

Why Docker?

Development and deployment with Docker, e.g. in a container isolated from your own system, have several advantages:

- Dependencies are installed only into the container, leaving your system clean.

- System setup is only done inside the container, leaving your system unaffected.

- Your application has a stable runtime environment with a reproducible way of rebuilding this environment.

- You can develop/deploy multiple applications with conflicting dependency sets/environment settings on one system - just provide a different container for each application.

- You can easily distribute your application as a built container image - and anyone can deploy it easily on any system that has Docker.

Terminology

- **Image** - a file system snapshot that can be "run" as a **container**

- **Container** - a running "lightweight virtual machine" that uses an **image** as its filesystem

It is important to understand that Docker uses layered images. E.g. one image is used as a *base image* and there can be many images built on top of it - each of them storing a different set of filesystem changes. When a container is run, Docker "squashes" the images, creating a single read-only filesystem for the container. All changes done in the container are recorded into a new image, that can be saved when container shuts down.

Docker Development/Deployment Workflow

This section summarizes development and deployment workflows for projects using Docker.

Dockerfile A crucial part of development/deployment workflow is Dockerfile. It's basically a list of instructions that says how to create an image for your application. It contains:

- Name of base image (this is usually obtained from Docker index of images).

- Zero or more shell commands that install dependencies/set up environment for the application

- List of exported ports (accessible from outside the container), mount points etc.

- A command that copies your application into the image.

Development Overally, the *development* workflow looks like this (assuming you have a *Dockerfile*):

- Build a fresh image.

- Instead of using source code that was copied into the image statically, mount the source directory from your system. This allows you to edit the code outside of the container (with your favourite editor/IDE), while running the code inside the container.

- Run the image to get a new container (this is actually done in one command with the previous step).

Rough equivalent of the above in Docker commandline invocations:

```
docker build -rm <dir> # <dir> is the directory containing Dockerfile
# -v mounts a local directory to the container, -P opens all ports specified in Dockerfile
docker run -v <local_path>:<container_path> -P <image_id>
```

Deployment The *deployment* workflow is quite similar:

- Build a fresh image (from a SCM revision that you want to distribute).

- Distribute the image.

This roughly translates to (if pushing to Docker index):

```
docker build <dir>
docker commit <container_id> myname/myapp
docker push myname/myapp
```

Implementation in DevAssistant

DevAssistant 0.9.0 comes with support for building Docker images and running Docker containers. Currently, the only assistant that supports creating new projects with Dockerfile is `crt python django`:

```
da create python django -n foo --docker
```

but we also have `mod docker develop` assistant, which is generally usable for any type of project that ships a Dockerfile. Use it like this:

```
da modify docker develop [-m MOUNTPOINT] [-i REUSE_IMAGE] [-p PATH]
```

If used with no arguments, this assistant searches for Dockerfile in current directory, builds a Docker image, mounts source code (the directory that contains Dockerfile) into it (mount point is determined based on first found ADD instruction in Dockerfile), runs a container and attaches to its output, so that you can develop and see the messages from process running inside the container.

By using the mentioned options, you can:

- override the directory where your sourcecode should be mounted (-m) in the container
- provide an image to use, if you've already built one (-i)
- specify path to your project if it's not in your current directory (-p)

1.1.2 A Brief Intro

DevAssistant is developer's best friend (right after coffee).

DevAssistant (http://devassistant.org) can help you with creating and setting up basic projects in various languages, installing dependencies, setting up environment etc. There are four main types of functionality provided:

- `da create` - create new project from scratch
- `da modify` - take local project and do something with it (e.g. import it to Eclipse)
- `da prepare` - prepare development environment for an upstream project or a custom task
- `da task` - perform a custom task not related to a specific project

The first three of these have shortcuts for faster use: "create" can be shortened as "crt", "modify" as "mod" and "prepare" as "prep".

DevAssistant is based on idea of per-{language/framework/...} "assistants" with hierarchical structure. E.g. you can run:

```
$ da create python django -n ~/myproject # sets up Django project named "myproject" inside your home
$ da create python flask -n ~/flaskproject # sets up Flask project named "flaskproject" inside your h
$ da create ruby rails -n ~/alsomyproject # sets up RoR project named "alsomyproject" inside your hom
```

DevAssistant also allows you to work with a previously created project, for example import it to Eclipse:

```
$ da modify eclipse # run in project dir or use -p to specify path
```

With DevAssistant, you can also prepare environment for developing upstream projects - either using project-specific assistants or using "custom" assistant for arbitrary projects (even those not created by DevAssistant):

```
$ da prepare custom custom -u scm_url
```

Warning: The custom assistant executes custom pieces of code from .devassistant file of the project. Therefore you have to be extra careful and use this **only with projects whose authors you trust**.

Last but not least, DevAssistant allows you to perform arbitrary tasks not related to a specific project:

```
$ da task <TODO:NOTHING YET>
```

Should you have some questions, feel free to ask us at Freenode channel #devassistant or on our mailing list (https://lists.fedoraproject.org/mailman/listinfo/devassistant). You can also join our G+ community (https://plus.google.com/u/0/communities/112692240128429771916) or follow us on Twitter (https://twitter.com/dev_assistant).

1.1.3 So What is an Assistant?

In short, assistant is a recipe for creating/modifying a project or setting up the environment in a certain way. DevAssistant is in fact just a core that "runs" assistants according to certain rules.

Each assistant specifies a way to achieve a single task, e.g. create a new project in framework X of language Y.

If you want to know more about how this all works, consult *Yaml Assistant Reference*.

Assistant Roles

There are four assistant roles:

creator (`create` or `crt` on command line) creates new projects

modifier (`modify` or `mod` on command line) works with existing projects

preparer (`prepare` or `prep` on command line) prepares environment for development of upstream projects

task (`task` on command line) performs arbitrary tasks not related to a specific project

The main purpose of having roles is separating different types of tasks. It would be confusing to have e.g. `python django` assistant (that creates new project) side-by-side with `eclipse` assistant (that registers existing project into Eclipse).

You can learn about how to invoke the respective roles below in *Creating New Projects*, *Modifying Existing Projects* and *Preparing Environment*.

1.1.4 Using Commandline Interface

Creating New Projects

DevAssistant can help you create (that's `crt` in the commands below) your projects with one line in a terminal. For example:

```
$ da create python django -n foo -e -g
```

`da` is the short form of `devassistant`. You can use either of them, but `da` is preferred.

This line will do the following:

- Install Django (RPM packaged) and all needed dependencies.

- Create a Django project named `foo` in the current working directory.

- Make any necessary adjustments so that you can run the project and start developing right away.

- The `-e` switch will make DevAssistant register the newly created projects into Eclipse (tries `~/workspace` by default, if you have any other, you need to specify it as an argument to `-e`). This will also cause installation of Eclipse and PyDev, unless already installed.

- The `-g` switch will make DevAssistant register the project on Github and push sources there. DevAssistant will ask you for your Github password the first time you're doing this and then it will create a Github API token and new SSH keys, so on any further invocation, this will be fully automatic. Note, that if your system username differs from your Github username, you must specify the Github username as an argument to `-g`.

Modifying Existing Projects

DevAssistant allows you to work with previously created projects. You can do this by using `da modify`, as opposed to `da create` for creating:

```
$ da modify eclipse
```

This will import a previously created project into Eclipse (and possibly install Eclipse and other dependencies implied by the project language). Optionally, you can pass `-p path/to/project` if your current working directory is not the project directory.

Preparing Environment

DevAssistant can set up the environment and install dependencies for development of an already existing project located in a remote SCM (e.g. Github). For custom projects you can use the `custom` assistant. Note that for projects that don't have `.devassistant` file, this will just checkout the sources:

```
$ da prepare custom -u scm_url
```

Warning: The `custom` assistant executes custom pieces of code from a `.devassistant` file, so use this only for projects whose upstreams you trust.

The plan is to also include assistants for well known and largely developed projects (that, of course, don't contain a `.devassistant` file). So in future you should be able to do something like:

```
$ da prepare openstack
```

and it should do everything needed to get you started developing OpenStack in a way that others do. But this is still somewhere in the future...

Tasks

The last piece of functionality is performing arbitrary tasks that are not related to a specific projects. E.g.:

```
$ da task <TODO:NOTHING YET>
```

Custom Actions

There are also some custom actions besides `crt`, `mod` and `prep`. For the time being, these are not of high importance, but in future, these will bring more functionality, such as making coffee for you.

help Displays help, what else?

version Displays current DevAssistant version.

1.1.5 Using the GUI

The DevAssistant GUI provides the full functionality of *Commandline Interface* through a Gtk based application.

As opposed to the CLI, which consists of three binaries, the GUI provides all assistant types (creating, modifying, preparing) in one, each type having its own page.

The GUI workflow is dead simple:

- Choose the assistant that you want to use, click it and possibly choose a proper subassistant (e.g. `django` for `python`).

- The GUI displays a window where you can modify some settings and choose from various assistant-specific options.

- Click the "Run" button and then just watch getting the stuff done. If your input is needed (such as confirming dependencies to install), DevAssistant will ask you, so don't go get your coffee just yet.

- After all is done, get your coffee and enjoy.

1.1.6 Currently Supported Assistants

Please note that list of currently supported assistants may vary greatly in different distributions, depending on available packages etc.

Currently supported assistants with their specialties (if any):

Creating

- C - a simple C project, allows you to create an SRPM and build an RPM by specifying `-b`

- C++

- Java - JSF - Java Server Faces project - Maven - A simple Apache Maven project

- Perl - Class - Simple class in Perl - Dancer - Dancer framework project

- PHP - LAMP - Apache/MySQL/PHP project

- Python - all Python assistants allow you to use `--venv` switch, which will make DevAssistant create a project inside a Python virtualenv and install dependencies there, rather then installing them system-wide from RPM - Django - Initial Django project, set up to be runnable right away - Flask - A minimal Flask project with a simple view and script for managing the application - Library - A custom Python library - PyGTK - Sample PyGTK project

- Ruby - Rails - Initial Ruby on Rails project

Modifying

- Eclipse - add an existing project into Eclipse (doesn't work for some languages/frameworks)

- Vim - install some interesting Vim extensions and make some changes in `.vimrc` (these changes will not affect your default configuration, instead you have to use the command `let devassistant=1` after invoking Vim)

Preparing

- Custom - checkout a custom previously created project from SCM (git only so far) and install needed dependencies

Tasks

<TODO: NOTHING YET>

1.2 Developer Documentation

1.2.1 DevAssistant Core

Note: So far, this only covers some bits and pieces of the whole core.

DevAssistant Load Paths

DevAssistant has couple of load path entries, that are searched for assistants, snippets, icons and files used by assistants. In standard installations, there are three paths:

1. "system" path, which is defined by OS distribution (usually `/usr/share/devassistant/`) or by Python installation (sth. like `/usr/share/pythonX.Y/devassistant/data/`)

2. "local" path, `/usr/local/share/devassistant/`

3. "user" path, `~/.devassistant/`

Another path(s) can be added by specifying `DEVASSISTANT_PATHS` environment variable (if more paths are used, they must be separated by colon). These paths are prepended to the list of standard load paths.

Each load path entry has this structure:

```
assistants/
  crt/
  mod/
  prep/
  task/
files/
  crt/
  mod/
  prep/
  task/
  snippets/
icons/
  crt/
  mod/
  prep/
  task/
snippets/
```

Icons under `icons` directory and files in `files` directory "copy" must the structure of `assistants` directory. E.g. for assistant `assistants/crt/foo/bar.yaml`, the icon must be `icons/crt/foo/bar.svg` and files must be placed under `files/crt/foo/bar/`

Assistants Loading Mechanism

DevAssistant loads assistants from all load paths mentioned above (more specifically from `<load_path>/assistants/` only), traversing them in order "system", "local", "user".

When DevAssistant starts up, it loads all assistants from all these paths. It assumes, that Creator assistants are located under `crt` subdirectories the same applies to Modifier (`mod`), Preparer (`prep`) and Task (`task`) assistants.

For example, loading process for Creator assistants looks like this:

1. Load all assistants located in `crt` subdirectories of each `<load path>/assistants/` (do not descend into subdirectories). If there are multiple assistants with the same name in different load paths, the first traversed wins.

2. For each assistant named `foo.yaml`:

   (a) If `crt/foo` directory doesn't exist in any load path entry, then this assistant is "leaf" and therefore can be directly used by users.

   (b) Else this assistant is not leaf and DevAssistant loads its subassistants from the directory, recursively going from point 1).

Command Runners

Command runners... well, they run commands. They are the functionality that makes DevAssistant powerful, since they effectively allow you to create callbacks to Python, where you can cope with the hard parts unsuitable for Yaml assistants.

When DevAssistant executes a `run` section, it reads commands one by one and dispatches them to their respective command runners. Every command runner can do whatever it wants - for example, we have a command runner that creates Github repos.

After a command runner is run, DevAssistant sets `LAST_LRES` and `LAST_RES` global variables for usage (these are rewritten with every command run). These variables represent the logical result of the command (`True`/`False`) and result (a "return value", something computed), much like with *Expressions*.

For reference of current commands, see *Command Reference*.

If you're missing some cool functionality, you can implement your own command runner and send us a pull request. (We're thinking of creating some sort of import hook that would allow assistants to import command runners from Python files outside of DevAssistant, but it's not on the priority list right now.) Each command must be a class with two classmethods:

```python
@register_command_runner
class MyCommandRunner(CommandRunner):
    @classmethod
    def matches(cls, c):
        return c.comm_type == 'mycomm'

    @classmethod
    def run(cls, c):
        input = c.input_res
        logger.info('MyCommandRunner was invoked: {ct}: {ci}'.format(ct=c.comm_type,
                                                                      ci=input))
        return (True, len(input))
```

This command runner will run all commands with command type `mycomm`. For example if your assistant contains:

```
run:
- $foo: $(echo "using DevAssistant")
- mycomm: You are $foo!
```

than DevAssistant will print out something like:

```
INFO: MyCommandRunner was invoked: mycomm: You are using DevAssistant!
```

When run, this command returns a tuple with *logical result* and *result*. This means you can assign the length of a string to a variable like this:

```
run:
$thiswillbetrue, $length~:
- mycomm: Some string.
```

(Also, `LAST_LRES` will be set to `True` and `LAST_RES` to length of the input string.)

Generally, the `matches` method should just decide (True/False) whether given command is runnable or not and the `run` method should actually run it. The `run` method should use `devassistant.logger.logger` object to log any messages and it can also raise any exception that's subclass of `devassistant.exceptions.ExecutionException`.

The `c` argument of both methods is a `devassistant.lang.Command` object. You can use various attributes of `Command`:

- **comm_type** - command type, e.g. `mycomm` (this will always be stripped of exec flag ~).

- **comm** - raw command input. The input is raw in the sense that it is uninterpreted. It's literally the same as what's written in assistant yaml file.

- **had_exec_flag** - `True` if the command type had exec flag, `False` otherwise.

- **input_log_res** and **input_res** - return values of input, see *Section Results*.

Note: input only gets evaluated one time - at time of using input_log_res or input_res. This means, among other things, that if exec flag is used, the command runner still has to access input_log_res or input_res to actually execute the input.

1.2.2 Tutorial: Creating Your Own Assistant

So you want to create your own assistant? There is nothing easier... They say that in all tutorials, right?

This tutorial will guide you through the process of creating simple assistants of *different roles* - Creator, Modifier, Preparer.

This tutorial doesn't cover everything. Consult *Yaml Assistant Reference* when you're missing something you really need to achieve. If you think that DevAssistant misses some functionality that would be useful, open a bug at https://www.github.com/bkabrda/devassistant/issues or send us a pull request.

Common Rules and Gotchas

Some things are common for all assistant types:

- Each assistant is one Yaml file, that must contain exactly one mapping - the so-called assistant attributes:

  ```
  fullname: My Assistant
  description: This will be part of help for this assistant
  ...
  ```

- You have to place them in a proper place, see *DevAssistant Load Paths* and *Assistants Loading Mechanism*.

- Files (e.g. templates, scripts, etc.) used by assistant should be placed in the same load dir, e.g. if your assistant is placed at `~/.devassistant/assistants`, DevAssistant will look for files under `~/.devassistant/files`.

- As mentioned in *DevAssistant Load Paths*, there are three main load paths in standard DevAssistant installation, "system", "local" and "user". The "system" dir is used for assistants delivered by your distribution/packaging system and you shouldn't touch or add files in this path. The "local" path can be used by system admins to add system-wide assistants while not touching "system" path. Lastly, "user" path can be used by users to create and use their own assistants. It is up to you where you place your assistant, but "user" path is usually best for playing around and development of new assistants. It is also the path that we will use throughout these tutorials.

Creating a Simple Creator

The title says it all. In this section, we will create a "Creator" assistant, that means an assistant that will take care of kickstarting a new project. We will write an assistant that creates a project containing a simple Python script that uses `argh` Python module. Let's suppose that we're writing this assistant for an RPM based system like Fedora, CentOS or RHEL.

This assistant is a "creator", so we have to put it somewhere into `~/.devassistant/assistants/crt/`. Since the standard DevAssistant distribution has a `python` assistant, it seems logical to make this new assistant a subassistant of `python`. That means that the assistant file will be `~/.devassistant/assistants/crt/python/argh.yaml`. It doesn't matter that the `python` assistant actually lives in a different load path, DevAssistant will hook the `argh` subassistant properly anyway.

Setting it Up

So, let's start writing our assistant by providing some initial metadata:

```
fullname: Argh Script Template
description: Create a template of simple script that uses argh library
project_type: [python]
```

If you now save the file and run `da create python argh -h`, you'll see that your assistant was already recognized by DevAssistant, although it doesn't provide any functionality yet. (Including project type in your Creator assistant is not necessary, but it may bring some benefits - see *Project Types*.

Dependencies

Now, we'll want to add a dependency on `python-argh` (which is how the package is called e.g. on Fedora). You can do this just by adding:

```
dependencies:
- rpm: [python-argh]
```

Now, if you save the file and actually try to run your assistant with `da create python argh`, it will install `python-argh`! (Well, assuming it's not already installed, in which case it will do nothing.) This is really supercool, but the assistant still doesn't do any project setup, so let's get on with it.

Files

Since we want the script to always look the same, we will create a file that our assistant will copy into proper place. This file should be put into `crt/python/argh` subdirectory the files directory (`~/.devassistant/files/crt/python/argh`). The file will be called `arghscript.py` and will have this content:

```python
#!/usr/bin/python2

from argh import *

def main():
    return 'Hello world'

dispatch_command(main)
```

We will need to refer to this file from our assistant, so let's open `argh.yaml` again and add a `files` section:

```
files:
  arghs: &arghs
    source: arghscript.py
```

DevAssistant will automatically search for this file in the correct directory, that is `~/.devassistant/files/crt/python/argh`. If there are e.g. some files common to multiple `python` subassistants, it is reasonable to place them into `~/.devassistant/files/crt/python` and refer to them with relative path like `../file.foo`

Run

Finally, we will be adding a `run` section, which is the section that does all the hard work. A `run` section is a list of **commands**. Every command is in fact a Yaml mapping with exactly one key and value. The key determines **command type**, while value is the **command input**. For example, `cl` is a **command type** that says that given **input** should be run on commandline, `log_i` is a **command type** that lets us print the **input** (message in this case) for user, etc.

Let's start writing our `run` section:

```
run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
```

But wait! We don't know what the project should be called and where it should be placed... Before we finish the `run` section, we'll need to add some arguments to our assistant.

Oh Wait, Arguments!

Creating any type of project typically requires some user input, at least name of the project to be created. To ask user for this sort of information, we can use DevAssistant arguments like this:

```
args:
  name:
    flags: [-n, --name]
    required: True
    help: 'Name of project to create'
```

This means that this assistant will have one argument called `name`. On commandline, it will expect `-n foo` or `--name foo` and since the argument is required, it will refuse to run without it.

You can now try running `da create python argh -h` and you'll see that the argument is printed out in commandline help.

Since there are some common arguments, the standard installation of DevAssistant ships with so called "snippets", that contain (among other things) definitions of frequentyl used arguments. You can use name argument for Creator assistants like this:

```
args:
  name:
    use: common_args
```

Note: up to version 0.8.0, "snippet" can also be used in place of "use"; "snippet" is obsolete and will be removed in 0.9.0.

Run Again

Now that we can obtain the desired name, let's continue. Now that we have the project name (let's assume that it's an arbitrary path to a directory where the argh script should be placed), we can continue. First, we will make sure that the

---

directory doesn't already exist. If so, we need to exit, because we don't want to overwrite or break something:

```
run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
- if $(test -e "$name"):
  - log_e: '"$name" already exists, can't proceed.'
```

There are few things to note here:

- There is a simple `if` condition with a shell command. If the shell command returns a non-zero value, the condition will evaluate to false, else it will evaluate to true. So in this case, if something exists at path `"$name"`, the condition will evaluate to true.

- In any command, we can use value of the `name` argument by prefixing argument name with `$` (so `$name` or `${name}`).

- The `log_e` command type is used to print a message and then abort the assistant execution immediately.

Let's continue by creating the directory. Add this line to `run` section:

```
- cl: mkdir -p "$name"
```

You may be wondering what will happen, if DevAssistant doesn't have write permissions or more generally if the `mkdir` command just fails. In this case, DevAssistant will exit, printing the output of failed command for user.

Next, we want to copy our script into the directory. We want to name it the same as name of the directory itself. But what if directory is a path, not simple name? We have to find out the project name and remember it somehow:

```
- $proj_name~: $(basename "$name")
```

What just happened? We assigned output of command `basename "$name"` to a new variable `proj_name` that we can use from now on. Note the `~` at the end of `$proj_name~`. This is called **execution flag** and it says that the command input should be executed as an expression, not taken as a literal. See *Expressions* for detailed expressions reference.

Note: the execution flag makes DevAssistant execute the input as a so-called "execution section". The input can either be a string, evaluated as an expression, or a list of commands, evaluated as another "run" section.

So let's copy the script and make it executable:

```
- cl: cp *arghs ${name}/${proj_name}.py
- cl: chmod +x ${name}/${proj_name}.py
```

One more thing to note here: by using `*arghs`, we reference a file from the `files` section.

Now, we'll use a super-special command:

```
- dda_c: "$name"
```

What is `dda_c`? The first part, `dda` stands for "dot devassistant file", the second part, `_c`, says, that we want to create this file (there are more things that can be done with `.devassistant` file, see *.devassistant Commands*). The "command" part of this call just says where the file should be stored, which is directory `$name` in our case.

The `.devassistant` file serves for storing meta information about the project. Amongst other things, it stores information about which assistant was invoked. This information can later serve to prepare the environment (e.g. install `python-argh`) on another machine. Assuming that we commit the project to a git repository, one just needs to run `da prepare custom -u <repo_url>`, and DevAssistant will checkout the project from git and use information stored in `.devassistant` to reinstall dependencies. (There is more to this, you can for example add a custom `run` section to `.devassistant` file or add custom dependencies, but this is not covered by this tutorial (not even by reference, so I need to place TODO here to document it).)

Note: There can be more dependencies sections and run sections in one assistant. To find out more about the rules of when they're used and how run sections can call each other, consult dependencies reference and run reference.

Something About Snippets

Wait, did we say git? Wouldn't it be nice if we could setup a git repository inside the project directory and do an initial commit? These things are always the same, which is exactly the type of task that DevAssistant should do for you.

Previously, we've seen usage of argument from snippet. But what if you could use a part of `run` section from there? Well, you can. And you're lucky, since there is a snippet called `git_init_add_commit`, which does exactly what we need. We'll use it like this:

```
- cl: cd "$name"
- use: git_init_add_commit.run
```

This calls section `run` from snippet `git_init_add_commit` in this place. Note, that all variables are "global" and the snippet will have access to them and will be able to change their values. However, variables defined in called snippet section will not propagate into current section.

Note: up to version 0.8.0, "call" can also be used in place of "use"; "call" is obsolete and will be removed in 0.9.0.

Finished!

It seems that everything is set. It's always nice to print a message that everything went well, so we'll do that and we're done:

```
- log_i: Project "$proj_name" has been created in "$name".
```

The Whole Assistant

... looks like this:

```
fullname: Argh Script Template
description: Create a template of simple script that uses argh library
project_type: [python]

dependencies:
- rpm: [python-argh]

files:
  arghs: &arghs
    source: arghscript.py

args:
  name:
    use: common_args

run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
- if $(test -e "$name"):
  - log_e: '"$name" already exists, cannot proceed.'
- cl: mkdir -p "$name"
- $proj_name~: $(basename "$name")
- cl: cp *arghs ${name}/${proj_name}.py
- cl: chmod +x *arghs ${name}/${proj_name}.py
- dda_c: "$name"
- cl: cd "$name"
- use: git_init_add_commit.run
- log_i: Project "$proj_name" has been created in "$name".
```

And can be run like this: `da create python argh -n foo/bar`.

Creating a Modifier

This section assumes that you've read the previous tutorial and are therefore familiar with DevAssistant basics. Modifiers are meant to modify existing projects, that means projects with `.devassistant` file (there is also an option to write assistant that modifies an arbitrary project without `.devassistant`, read on).

Modifier Specialties

The special behaviour of modifiers only applies if you use dda_r in pre_run section. This command reads .devassistant file from given directory and puts the read variables in global variable context, so they're available from all the following dependencies and run section.

If modifier reads `.devassistant` file in `pre_run` section, DevAssistant tries to search for more `dependencies` sections to use. If the project was previously created by `crt python django`, the engine will install dependencies from sections `dependencies_python_django`, `dependencies_python` and `dependencies`.

Also, the engine will try to run `run_python_django` section first, then it will try `run_python` and then `run` - note, that this will only run the first found section and then exit, unlike with dependencies, where all found sections are used.

– IN PROGRESS –

1.2.3 Packaging Your Assistant

Note: this functionality is under heavy development and is not fully implemented yet.

So now you know how to *create an assistant*. But what if you want to share your assistant with others?

For that you could send them all the files from your assistant and tell them where they belong. But that would be very unpleasant and that's why we've invented dap. Dap is a format of extension for DevAssistant that contains custom assistants. It means DevAssistant Package.

A dap is a tar.gz archive with `.dap` extension. The name of a dap is always `<package_name>-<version>.dap` - i.e. `foo-0.0.1.dap`.

Directory structure of a dap

The directory structure of a dap copies the structure of `~/.devassistant` or `/usr/share/devassistant` folder. The only difference is, that it can only contain assistants, files and icons that that belongs to it's namespace.

Each dap has an unique name (lat's say `foo`) and it can only contain assistants `foo` or `foo/*`. Therefore, the directory structure looks like this:

```
foo-0.0.1/
  meta.yaml
  assistants/
    {crt,mod,prep,task}/
      foo.yaml
      foo/
  files/
    {crt,mod,prep,task,snippets}/
      foo/
  snippets/
```

```
      foo/
  icons/
    {crt,mod,prep,task,snippets}/
      foo.{png,svg}
      foo/
  doc/
      foo/
```

Note several things:

- Each of this is optional, i.e. you don't create `files` or `snippets` folder if you provide no files or snippets. Only mandatory thing is `meta.yaml` (see below).

- Everything goes to the particular folder, just like you've learn in the *Tutorial*. However, you can only add stuff named as your dap (means either a folder or a file with a particular extension). If you have more levels of assistants, such as `crt/foo/bar/spam.yaml`, you have to include top-level assistants (in this case both `crt/foo.yaml` and `crt/foo/bar.yaml`). And you have to preserve the structure in other folders as well (i.e. no `icons/crt/foo/spam.svg` but `icons/crt/foo/bar/spam.svg`).

- The top level folder is named `<package_name>-<version>`.

meta.yaml

```
package_name: foo # required
version: 0.0.1 # required
license: GPLv2 # required
authors: [Bohuslav Kabrda <bkabrda@mailserver.com>, ...] # required
homepage: https://github.com/bkabrda/assistant-foo # optional
summary: Some brief one line text # required
bugreports: <a single URL or email address> # optional
description: |
    Some not-so-brief optional text.
    It can be split to multiple lines.

    BTW you can use **Markdown**.
```

- **package name** can contain lowercase letters (ASCII only), numbers, underscore and dash (while it can only start and end with a letter or digit), it has to be unique, several names are reserved by DevAssitant itself (e.g. python, ruby)

- **version** follows this scheme: <num>[.<num>]*[dev|a|b], where 1.0.5 < 1.1dev < 1.1a < 1.1b < 1.1

- **license** is specified via license tag used in Fedora https://fedoraproject.org/wiki/Licensing:Main?rd=Licensing#Good_Licenses

- **authors** is a list of authors with their e-mail addresses (_at_ can be used instead of @)

- **homepage** is an URL to existing webpage that describes the dap or contains the code (such as in example), only http(s) or ftp is allowed, no IP addresses

- **summary** and **description** are self-descriptive in the given example

- **bugreports** defines where the user should report bugs, it can be either an URL (issue tracker) or an e-mail address (mailing list or personal)

Checking your dap for sanity

Once you have your dap packaged, check it for sanity with `daplint` tool from `daploader`.

First, you have to get the `daplint` tool. Install daploader with `pip` or `easy_install`.

```
pip install daploader
```

Then you can check your dap with `daplint`:

```
daplint foo-0.0.1.dap
```

Uploading your dap to DevAssistant Package Index

When you are satisfied, you can share your assistant on Dapi (DevAssistant Package Index).

On Dapi, log in with Github or Fedora account and follow Upload a Dap link in the menu.

1.2.4 Yaml Assistant Reference

Note: The Yaml DSL has changed significantly in 0.9.0 in backwards incompatible manner. This documentation is only for version 0.9.0 and later.

This is a reference manual to writing yaml assistants. Yaml assistants use a special DSL defined on this page. For real examples, have a look at assistants in our Github repo.

Why the hell another DSL? When we started creating DevAssistant and we were asking people who work in various languages whether they'd consider contributing assistants for those languages, we hit the "I'm not touching Python" barrier. Since we wanted to keep the assistants consistent (centralized logging, sharing common functionality, same backtraces, etc...), we created a new DSL. So now we have something that everyone complains about, including Pythonists, which seems to be consistent too ;)

Assistant Roles

For list and description of assistant roles see *Assistant Roles*.

The role is implied by assistant location in one of the load path directories, as mentioned in *Assistants Loading Mechanism*.

All the rules mentioned in this document apply to all types of assistants, with exception of sections *Modifier Assistants*, *Preparer Assistants* and *Task Assistants* that talk about specifics of Modifier, resp. Preparer, resp. Task assistants.

Assistant Name

Assistant name is a short name used on command line, e.g. `python`. Historically, it had to be the only top-level yaml mapping in the file, e.g.:

```
python:
  fullname: Python
  description: Some verbose description
```

Since DevAssistant 0.9.0, it is preferred to omit it and just provide the assistant attributes as the top level mapping:

```
fullname: Python
description: Some verbose description
```

Assistant name is derived from the filename by stripping the `.yaml` extension, e.g. assistant `python.yaml` file is named `python`.

Assistant Attributes

Assistant attributes form the top level mapping in Yaml file:

```
fullname: Python

run:
- cl: mkdir -p $name
- log_i: I'm in $name
```

List of allowed attributes follows (all of them are optional, and have some sort of reasonable default, it's up to your consideration which of them to use):

fullname a verbose name that will be displayed to user (`Python Assistant`)

description a (verbose) description to show to user (`Bla bla create project bla bla`)

dependencies (and **dependencies_*)** specification of dependencies, see below Dependencies

args specification of arguments, see below Args

files specification of used files, see below Files

project_type type of the project, see *Project Types*

run (and **run_*)** specification of actual operations, see *Run Sections Reference*

pre_run and post_run specification of operations to carry out before/after running main `run` section, see below Assistants Invocation; follow the rules specified in *Run Sections Reference*

files_dir directory where to take files (templates, helper scripts, ...) from. Defaults to base directory from where this assistant is taken + `files`. E.g. if this assistant is `~/.devassistant/assistants/crt/path/and/more.yaml`, files will be taken from `~/.devassistant/files/crt/path/and/more` by default.

icon_path absolute or relative path to icon of this assistant (will be used by GUI). If not present, a default path will be used - this is derived from absolute assistant path by replacing `assistants` by `icons` and `.yaml` by `.svg` - e.g. for `~/.devassistant/assistants/crt/foo/bar.yaml`, the default icon path is `~/.devassistant/icons/crt/foo/bar.svg`

Assistants Invocation

When you invoke DevAssistant with it will run following assistants sections in following order:

- `pre_run`
- `dependencies`
- `run` (possibly different section for Modifier Assistants)
- `post_run`

If any of the first three sections fails in any step, DevAssistant will immediately skip to `post_run` and the whole invocation will be considered as failed (will return non-zero code on command line and show "Failed" in GUI).

Dependencies

Yaml assistants can express their dependencies in multiple sections.

- Packages from section `dependencies` are **always** installed.

- If there is a section named `dependencies_foo`, then dependencies from this section are installed **iff** `foo` argument is used (either via commandline or via gui). For example:

  ```
  $ da python --foo
  ```

- These rules differ for Modifier Assistants

Each section contains a list of mappings `dependency type: [list, of, deps]`. If you provide more mappings like this:

```
dependencies:
- rpm: [foo]
- rpm: ["@bar"]
```

they will be traversed and installed one by one. Supported dependency types:

rpm the dependency list can contain RPM packages or YUM groups (groups must begin with `@` and be quoted, e.g. `"@Group name"`)

use / call (these two do completely same, call is obsolete and will be removed in 0.9.0) installs dependencies from snippet/another dependency section of this assistant/dependency section of superassistant. For example:

```
dependencies:
- use: foo.dependencies
- use: foo.dependencies_bar # will install dependencies from snippet "foo", section "bar"
- use: self.dependencies_baz # will install dependencies from section "dependencies_baz" of this
- use: super.dependencies # will install dependencies from "dependencies" section of first super
```

if, else conditional dependency installation. For more info on conditions see *Run Sections Reference*. A very simple example:

```
dependencies:
- if $foo:
  - rpm: [bar]
- else:
  - rpm: [spam]
```

Full example:

```
dependencies: - rpm: [foo, "@bar"]

dependencies_spam:
- rpm: [beans, eggs]
- if $with_spam:
  - use: spam.spamspam
- rpm: ["ham${more_ham}"]
```

Sometimes your dependencies may get terribly complex - they depend on many parameters, you need to use them dynamically during `run`, etc. In these cases, consider using *Dependencies Command* in `run` section.

Args

Arguments are used for specifying commandline arguments or gui inputs. Every assistant can have zero to multiple arguments.

The `args` section of each yaml assistant is a mapping of arguments to their attributes:

```
args:
  name:
    flags:
    - -n
    - --name
  help: Name of the project to create.
```

Available argument attributes:

flags specifies commandline flags to use for this argument. The longer flag (without the `--`, e.g. `name` from `--name`) will hold the specified commandline/gui value during `run` section, e.g. will be accessible as `$name`.

help a help string

required one of `{true,false}` - is this argument required?

nargs how many parameters this argument accepts, one of `{0, ?,*,+}` (e.g. {0, 0 or 1, 0 or more, 1 or more})

default a default value (this will cause the default value to be set even if the parameter wasn't used by user)

action one of `{store_true, [default_iff_used, value]}` - the `store_true` value will create a switch from the argument, so it won't accept any parameters; the `[default_iff_used, value]` will cause the argument to be set to default value `value` **iff** it was used without parameters (if it wasn't used, it won't be defined at all)

use / snippet (these two do completely same, snippet is obsolete and will be removed in 0.9.0) name of the snippet to load this argument from; any other specified attributes will override those from the snippet By convention, some arguments should be common to all or most of the assistants. See *Common Assistant Behaviour*

preserved if set, the value of this argument will be saved and will reappear in the next launch of devassistant GUI. The attribute string is a key under which the argument value will be stored. The key should be of the form "scope.argname" so that you can either share the value across more assistants or avoid collisions if any other assistant uses an argument with same name but different meaning. The argument values are stored in "~/.devassistant/.config". It is ignored in command-line interface.

Gui Hints

GUI needs to work with arguments dynamically, choose proper widgets and offer sensible default values to user. These are not always automatically retrieveable from arguments that suffice for commandline. For example, GUI cannot meaningfully prefill argument that says it "defaults to current working directory". Also, it cannot tell whether to choose a widget for path (with the "Browse ..." button) or just a plain text field.

Because of that, each argument can have `gui_hints` attribute. This can specify that this argument is of certain type (path/str/bool) and has a certain default. If not specified in `gui_hints`, the default is taken from the argument itself, if not even there, a sensible "empty" default value is used (home directory/empty string/false). For example:

```
args:
  path:
    flags:
    - [-p, --path]
    gui_hints:
      type: path
      default: $(pwd)/foo
```

If you want your assistant to work properly with GUI, it is good to use `gui_hints` (currently, it only makes sense to use it for `path` attributes, as `str` and `bool` get proper widgets and default values automatically).

Files

This section serves as a list of aliases of files stored in one of the `files` dirs of DevAssistant. E.g. if your assistant is `assistants/crt/foo/bar.yaml`, then files are taken relative to `files/crt/foo/bar/` directory. So if you have a file `files/crt/foo/bar/spam`, you can use:

```
files:
  spam: &spam
    source: spam
```

This will allow you to reference the `spam` file in `run` section as `*spam` without having to know where exactly it is located in your installation of DevAssistant.

Run

Reference for run sections has a separate page: *Run Sections Reference*.

Modifier Assistants

Modifier assistants are assistants that are supposed to work with already created project. They must be placed under `mod` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*.

There are few special things about modifier assistants:

- They usually utilize `dda_r` to read the whole `.devassistant` file (usually from directory specified by `path` variable or from current directory). *Since version 0.8.0, every modifier assistant has to do this on its own, be it in pre_run or run section*. This also allows you to modify non-devassistant projects - just don't use `dda_r`.

The special rules below *only apply if you use dda_t in pre_run section*.

- They use dependency sections according to the normal rules + they use *all* the sections that are named according to `project_type` loaded from `.devassistant`, e.g. if `project_type` is `[foo, bar]`, dependency sections `dependencies`, `dependencies_foo` and `dependencies_foo_bar` will be used as well as any sections that would get installed according to specified parameters. The rationale behind this is, that if you have e.g. `eclipse` modifier that should work for both `python django` and `python flask` projects, chance is that they have some common dependencies, e.g. `eclipse-pydev`. So you can just place these common dependencies in `dependencies_python` and you're done (you can possibly place special per-framework dependencies into e.g. `dependencies_python_django`).

- By default, they don't use `run` section. Assuming that `project_type` is `[foo, bar]`, they first try to find `run_foo_bar`, then `run_foo` and then just `run`. The first found is used. If you however use cli/gui parameter `spam` and section `run_spam` is present, then this is run instead.

Preparer Assistants

Preparer assistants are assistants that are supposed to checkout sources of upstream projects and set up environment for them (possibly utilizing their `.devassistant` file, if they have one). Preparers must be placed under `prep` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*.

Preparer assistants commonly utilize the `dda_dependencies` and `dda_run` commands in `run` section.

Task Assistants

Task assistants are supposed to carry out arbitrary task that are not related to a specific project. <TODO>

1.2.5 Run Sections Reference

Run sections are the essence of DevAssistant. They are responsible for performing all the tasks and actions to set up the environment and the project itself. For Creator and Preparer assistants, the section named `run` is always invoked, *Modifier Assistants* may invoke different sections based on metadata in a `.devassistant` file.

Note, that `pre_run` and `post_run` follow the same rules as `run` sections. See *Assistants Invocation* to find out how and when these sections are invoked.

Every section is a sequence of various **commands**, mostly invocations of commandline. Each command is a mapping of **command type** to **command input**:

```
run:
- command_runner: command_input
- command_runner_2: another_command_input
```

Note, that **section** is a general term used for any sequence of commands. Sections can have subsections (e.g. in conditions or loops), assuming they follow some rules (see below).

Introduction to Commands and Variables

The list of all supported commands can be found at *Command Reference*, we only document the basic usage of the most important commands here. Note, that when you use variables (e.g. `$variable`) in command input, they get substituted for their values (undefined variables will remain unchanged).

- command line invocation:

  ```
  - cl: mkdir -p $spam
  ```

  This will invoke a subshell and create a directory named `$spam`. If the command returns non-zero return code, DevAssistant will fail.

- logging:

  ```
  - log_i: Directory $spam created.
  ```

  This command will log the given message at `INFO` level - either to terminal or GUI. You can use similar commands to log at different log levels: `log_d` for `DEBUG`, `log_w` for `WARNING`, `log_e` for `ERROR` and `log_c` for `CRITICAL`. By default, messages of level `INFO` and higher are logged. Log messages with levels `ERROR` and `CRITICAL` emit the message and then terminate execution of DevAssistant immediately.

- conditions:

  ```
  - if not $foo and $(ls /spam/spam/spam):
    - log_i: This gets executed if the condition is satisfied.
  - else:
    - log_i: Else this section gets executed.
  ```

  Conditions work as you'd expect in any programming language - `if` subsection gets executed if the condition evaluates to true, otherwise `else` subsection gets executed. The condition itself is an **expression**, see *Expressions* for detailed reference of expressions.

- loops:

  ```
  - for $i word_in $(ls):
    - log_i: Found file $i.
  ```

  Loops probably also work as you'd expect - they've got the control variable and an iterable. Loop iterators are **expressions**, see *Expressions*. Note, that you can use two forms of for loop. If you use `word_in`, DevAssistant

will split the given expression on whitespace and then iterate over that, while if you use `in`, DevAssistant will iterate over single characters of the string.

- variable assignment:

  - `$foo: "Some literal with value of "foo" variable: $foo"`

  This shows how to assign a literal value to a variable. It is also possible to assign the result of another command to a variable, see Section Results for how to use the execution flag.

Remember to check *Command Reference* for a comprehensive description of all commands.

Literal Sections vs. Execution Sections

DevAssistant distinguishes two different section types: **input sections** and **execution sections**. Some sections are inherently execution sections:

- all `run` sections of assistants
- `if`, `else` subsections
- `for` subsections

Generally, execution sections can be either:

- *expression* (e.g. a Yaml string that gets interpreted as an expression)

or

- section (sequence of **commands**)

Literal section can be any valid Yaml structure - string, list or mapping.

Section Results

Similarly to *expressions*, sections return *logical result* and *result*:

- literal section

  - *result* is a string/list/mapping with variables substituted for their values
  - *logical result* is False if the structure is empty (empty string, list or mapping), True otherwise

- execution sections

  - *result* is the result of last command of given section
  - *logical result* is the logical result of last command of given section

Some examples follow:

```
run:
# now we're inherently in an execution section
- if $(ls /foo):
  # now we're also in an execution section, e.g. the below sequence is executed
  - foo:
      # the input passed to "foo" command runner is inherently a literal input, e.g. not executed
      # this means foo command runner will get a mapping with two key-value pairs as input, e.g.:
      # {'some': 'string value', 'with': [ ... ]}
      some: string value
      with: [$list, $of, $substituted, $variables]
- $var: this string gets assigned to "var" with $substituted $variables
```

If you need to assign the result of an expression or execution section to a variable or pass it to a command runner, you need to use the **execution flag**: ~:

```
run:
- $foo~: ($this or $gets) and $executed_as_expression
- foo~:
  # input of "foo" command runner will be result of the below execution section
  - command_runner: literal_section
  - command_runner_2~:
    # similarly, input of command_runner_2 will be result of the below execution section
    - cr: ci
    - cr2: ci2
```

Note, that a string starting with the execution flag is also executed as an expression. If you want to create a literal that starts with ~, just use the escape value for it (~~):

```
run:
- $foo: ~$(ls) and $bar
- $bar: ~~/some_dir_in_users_home
- log_i: The tilde character (~) only needs to be escaped when starting a string.
```

Each command specifies its return value in a different way, see *Command Reference*.

Variables Explained

Initially, variables are populated with values of arguments from the commandline/gui and there are no other variables defined for creator assistants. For modifier assistants global variables are prepopulated with some values read from .devassistant. You can either define (and assign to) your own variables or change the values of current ones.

Additionally, after each command, variables $LAST_RES and $LAST_LRES are populated with the result of the last command (these are also the return values of the command) - see *Command Reference*

The variable scope works as follows:

- When invoking a different run section (from the current assistant or snippet), the variables get passed by value (e.g. they don't get modified for the remainder of this scope).

- Variables defined in subsections (if, else, for) continue to be available until the end of the current run section.

All variables are global in the sense that if you call a snippet or another section, it can see all the arguments that are defined.

Quoting

When using variables that contain user input, they should always be quoted in the places where they are used for bash execution. That includes cl* commands, conditions that use bash return values and variable assignment that uses bash.

Global Variables

In all assistants, a few useful global variables are available. These include:

- $__system_name__ - name of the system, e.g. "linux"

- $__system_version__ - version of the system, e.g. "3.13.3-201.fc20.x86_64"

- $__distro_name__ - name of Linux distro, e.g. "fedora"

- `$__distro_version__` - version of Linux distro, e.g. "20"

Note: if any of this information is not available, the corresponding variable will be empty. Also note, that you can rely on all the variables having lowercase content.

Expressions

Expressions are used in assignments, conditions and as loop "iterables". Every expression has a *logical result* (meaning success - `True` or failure - `False`) and *result* (meaning output). *Logical result* is used in conditions and variable assignments, *result* is used in variable assignments and loops. Note: when assigned to a variable, the *logical result* of an expression can be used in conditions as expected; the *result* is either `True` or `False`.

Syntax and semantics:

- `$foo`
    - if `$foo` is defined:
        * *logical result*: `True` *iff* value is not empty and it is not `False`
        * *result*: value of `$foo`
    - otherwise:
        * *logical result*: `False`
        * *result*: empty string
    - *note*: boolean values (e.g. those acquired by argument with `action:  store_true`) always have an empty string as a *result* and their value as *logical result*
- `$(commandline command)` (yes, that is a command invocation that looks like running command in a subshell)
    - if `commandline command` has return value 0:
        * *logical result*: `True`
    - otherwise:
        * *logical result*: `False`
    - regardless of *logical result*, *result* always contains both stdout and stderr lines in the order they were printed by `commandline command`
- `as_root $(commandline command)` runs `commandline command` as superuser; DevAssistant may achieve this differently on different platforms, so the actual way how this is done is considered to be an implementation detail
- `defined $foo` - works exactly as `$foo`, but has *logical result* `True` even if the value is empty or `False`
- `not $foo` negates the *logical result* of an expression, while leaving *result* intact
- `$foo and $bar`
    - *logical result* is the logical conjunction of the two arguments
    - *result* is an empty string if at least one of the arguments is empty, or the latter argument
- `$foo or $bar`
    - *logical result* is the logical disjunction of the two arguments
    - *result* is the first non-empty argument or an empty string
- `literals` – `"foo"`, `'foo'`

- *logical result* `True` for non-empty strings, `False` otherwise

- *result* is the string itself, sans quotes

- *Note: If you use an expression that is formed by just a literal, e.g.* `"foo"` *, then DevAssistant will fail, since Yaml parser will strip these. Therefore you have to use* `'"foo"'` *.*

- `$foo in $bar`

  - *logical result* is `True` if the result of the second argument contains the result of the second argument (e.g. "inus" in "Linus Torvalds") and `False` otherwise

  - *result* is always the first agument

All these can be chained together, so, for instance, `"1.8.1.4" in $(git --version) and defined $git` is also a valid expression

1.2.6 Command Reference

This page serves as a reference for commands of the DevAssistant Yaml DSL. Every command consists of a **command_type** and **command_input**. After it gets executed, it sets the `LAST_LRES` and `LAST_RES` variables. These are also its return values, similar to *Expressions* **logical result** and **result**.

- `LAST_LRES` is the logical result of the run - `True`/`False` if successful/unsuccessful

- `LAST_RES` is the "return value" - e.g. a computed value

In the Yaml DSL, commands are called like this:

```
command_type: command_input
```

This reference summarizes commands included in DevAssistant itself in the following format:

`command_type` - some optional info

- Input: what should the input look like?

- RES: what is `LAST_RES` set to after this command?

- LRES: what is `LAST_LRES` set to after this command?

- Example: example usage

Missing something? Commands are your entry point for extending DevAssistant. If you're missing some functionality in `run` sections, just *write a command runner* and send us a pull request.

Builtin Commands

There are three builtin commands that are inherent part of DevAssistant Yaml DSL:

- variable assignment

- condition

- loop

All of these builtin commands utilize expressions in some way - these must follow rules in *Expressions*.

Variable Assignment

Assign *result* (and possibly also *logical result*) of *Expressions* to a variable(s).

`$<var1>[, $<var2>]` - if one variable is given, *result* of expression (**command input**) is assigned. If two variables are given, the first gets assigned *logical result* and the second *result*.

- Input: an expression

- RES: *result* of the expression

- LRES: *logical result* of the expression

- Example:

```
$foo: "bar"
$spam:
- spam
- spam
- spam
$bar: $baz
$success, $list~: $(ls "$foo")
```

Condition

Conditional execution.

`if <expression>`, `else` - conditionally execute one or the other section (`if` can stand alone, of course)

- Input: a subsection to run

- RES: RES of last command in the subsection, if this clause is invoked. If not invoked, RES remains untouched.

- LRES: LRES of last command in the subsection, if this clause is invoked. If not invoked, LRES remains untouched.

- Example:

```
if defined $foo:
- log_i: Foo is $foo!
else:
- log_i: Foo is not defined!
```

Loop

A simple for loop.

`for <var>[, <var>] [word_in,in] <expression>` - loop over result of the expression. If `word_in` is used and `<expression>` is a string, it will be split on whitespaces and iterated over; with `in`, string will be split to single characters and iterated over. For iterations over lists and mappings, `word_in` and `in` behave the same. When iterating over mapping, two control variables may be provided to get both key and its value.

- Input: a subsection to repeat in loop

- RES: RES of last command of last iteration in the subsection. If there are no interations, RES is untouched.

- LRES: LRES of last command of last iteration in the subsection. If there are no interations, RES remains untouched.

- Example:

```
for $i word_in $(ls):
- log_i: File: $i

$foo:
  1: one
  2: two
for $k, $v in $foo:
- log_i: $k, $v
```

Ask Commands

User interaction commands, let you ask for password and various other input.

`ask_confirm`

- Input: mapping containing `prompt` (short prompt for user) and `message` (a longer description of what the user should confirm)

- RES: the confirmation (`True` or `False`)

- LRES: same as RES

- Example:

```
- $confirmed~:
  - ask_confirm:
      message: "Do you think DevAssistant is great?"
      prompt: "Please select yes."
```

`ask_input`

- Input: mapping containing `prompt` (short prompt for user) and optionally `message` (a longer description)

- RES: the string that was entered by the user

- LRES: `True` if non-empty string was provided

- Example:

```
- $variable:
  - ask_input:
      prompt: "Your name"
```

`ask_password`

- Input: mapping containing `prompt` (short prompt for user)

- This command works the same way as `ask_input`, but the entered text is hidden (displayed as bullets)

- RES: the password

- LRES: `True` if non-empty password was provided

- Example:

```
- $passwd:
  - ask_password:
      prompt: "Please provide your password"
```

Command Line Commands

Run commands in subprocesses and receive their output.

cl, cl_[i,r] (these do the same, but appending i logs the command output on INFO level and appending r runs command as root)

- Input: a string, possibly containing variables and references to files

- RES: stdout + stdin interleaved as they were returned by the executed process

- LRES: always True (if the command fails, the whole DevAssistant execution fails)

- Example:

```
cl: mkdir ${name}
cl: cp *file ${name}/foo
cl_i: echo "Hey!"
cl_ir: echo "Echoing this as root"
cl_r: mkdir /var/lib/foo
```

Note: when using r, it's job of DevAssistant core to figure out what to use as authentication method. Consider this an implementation detail.

A note on changing current working directory: Due to the way Python interpreter works, DevAssistant has to special-case "cd <dir>" command, since it needs to call a special Python method for changing current working directory of the running interpreter. Therefore you must always use "cd <dir>" as a single command (do not use "ls foo && cd foo"); also, using pushd/popd is not supported for now.

Dependencies Command

Install dependencies from given **command input**.

dependencies

- Input: list of mappings, similar to *Dependencies section*, but without conditions and usage of sections from snippets etc.

- RES: **command input**, but with expanded variables

- LRES: always True (terminates DevAssistant if dependency installation fails)

- Example:

```
if $foo:
- $rpmdeps: [foo, bar]
else:
- $rpmdeps: []

dependencies:
- rpm: $rpmdeps
```

.devassistant Commands

Commands that operate with .devassistant file.

dda_c - creates a .devassistant file, should only be used in creator assistants

- Input: directory where the file is supposed to be created

- RES: always True, terminates DevAssistant if something goes wrong

---

- LRES: always empty string

- Example:

  ```
  dda_c: ${path}/to/project
  ```

`dda_r` - reads an existing `.devassistant` file, should be used by modifier and preparer assistants.Sets some global variables accordingly, most importantly `original_kwargs` (arguments used when the project was created) - these are also made available with `dda__` prefix (yes, that's double underscore).

- Input: directory where the file is supposed to be

- RES: always empty string

- LRES: always `True`, terminates DevAssistant if something goes wrong

- Example:

  ```
  dda_r: ${path}/to/project
  ```

`dda_w` - writes a mapping (dict in Python terms) to `.devassistant`

- Input: list with directory with `.devassistant` file as a first item and the mapping to write as the second item. Variables in the mapping will be substituted, you have to use `$$foo` (two dollars instead of one) to get them as variables in `.devassistant`.

- RES: always empty string

- LRES: always `True`, terminates DevAssistant if something goes wrong

- Example:

  ```
  dda_w:
  - ${path}/to/project
  - run:
    - $$foo: $name # name will get substituted from current variable
    - log_i: $$foo
  ```

`dda_dependencies` - installs dependencies from `.devassistant` file, should be used by preparer assistants. Utilizes both dependencies of creator assistants that created this project plus dependencies from `dependencies` section, if present (this section is evaluated in the context of current assistant, not the creator).

- Input: directory where the file is supposed to be

- RES: always empty string

- LRES: always `True`, terminates DevAssistant if something goes wrong

- Example:

  ```
  dda_dependencies: ${path}/to/project
  ```

`dda_run` - run `run` section from from `.devassistant` file, should be used by preparer assistants. This section is evaluated in the context of current assistant, not the creator.

- Input: directory where the file is supposed to be

- RES: always empty string

- LRES: always `True`, terminates DevAssistant if something goes wrong

- Example:

  ```
  dda_run: ${path}/to/project
  ```

Github Command

Manipulate Github repositories.

Github command (`github`) has many "subcommands". Subcommands are part of the command input, see below.

- Input: a string with a subcommand or a two item list, where the first item is a subcommand and the second item is a mapping that explicitly specifies parameters for the subcommand.

- RES: if command succeeds, either a string with URL of manipulated repo or empty string is returned (depends on subcommand), else a string with problem description (it is already logged at WARNING level)

- LRES: `True` if the Github operation succeeds, `False` otherwise

- Example:

```
github: create_repo

github:
- create_and_push
- login: bkabrda
  reponame: devassistant

github: push

github: create_fork
```

Explanation of individual subcommands follows. Each subcommand takes defined arguments, whose default values are taken from global context. E.g. `create_and_push` takes an argument `login`. If it is not specified, assistant variable `github` is used.

create_repo Creates a repo with given `reponame` (defaults to var `name`) for a user with given login (defaults to var `github`). Optionally accepts `private` argument to create repo as private (defaults to var `github_private`).

create_and_push Same as `create_repo`, but it also adds a proper git remote to repository in current working dir and pushes to Github.

push Just does `git push -u origin master`, no arguments needed.

create_fork Creates a fork of repo at given `repo_url` (defaults ot var `url`) under user specified by `login` (defaults to var `github`).

Jinja2 Render Command

Render a Jinja2 template.

`jinja_render`, `jinja_render_dir` - render a single template or a directory containing more templates

- Input: a mapping containing

  - `template` - a reference to file (or a directory if using `jinja_render_dir`) in `files` section

  - `destination` - directory where to place rendered template (or rendered directory)

  - `data` - a mapping of values used to render the template itself

  - `overwrite` (optional) - overwrite the file if it exists? (defaults to `false`)

  - `output` (optional) - specify a filename of the rendered template (see below for information on how the filename is costructed if not provided), not used with `jinja_render_dir`

- RES: always `success` string

- LRES: always `True`, terminates DevAssistant if something goes wrong

- Example:

```
jinja_render:
  template: *somefile
  destination: ${dest}/foo
  overwrite: yes
  output: filename.foo
  data:
    foo: bar
    spam: spam

jinja_render_dir:
  template: *somedir
  destination: ${dest}/somedir
  data:
    foo: foo!
    spam: my_spam
```

The filename of the rendered template is created in this way (the first step is omitted with `jinja_render_dir`:

- if `output` is provided, use that as the filename

- else if name of the template endswith `.tpl`, strip `.tpl` and use it

- else use the template name

For template syntax reference, see Jinja2 documentation.

Logging Commands

Log commands on various levels. Logging on ERROR or CRITICAL logs the message and then terminates the execution.

`log_[d,i,w,e,c]` (the letters stand for DEBUG, INFO, WARNING, ERROR, CRITICAL)

- Input: a string, possibly containing variables and references to files

- RES: the logged message (with expanded variables and files)

- LRES: always `True`

- Example:

```
log_i: Hello $name!
log_e: Yay, something has gone wrong, exiting.
```

> **Warning:** If you start your log command with an apostrophe or a quotation mark, you must end the line with the same character, and it must not appear elsewhere on the line

SCL Command

Run subsection in SCL environment.

`scl [args to scl command]` (note: you **must** use the scriptlet name - usually `enable` - because it might vary)

- Input: a subsection

- RES: RES of the last command in the given section

- LRES: LRES of the last command in the given section

- Example:

```
– scl enable python33 postgresql92:
  – cl_i: python --version
  – cl_i: pgsql --version
```

Note: currently, this command can't be nested, e.g. you can't run `scl enable` in another `scl enable`.

Running Commands as Another User

Run subsection as a different user (how this command runner does this is considered an implementation detail). `as <username>` (note: use `as root`, to run subsection under superuser)

- Input: a subsection

- RES: output of **the whole** subsection

- LRES: LRES of the last command in the given section

- Example:

```
– as root:
  – cl: ls /root
– as joe:
  – log_i~: $(echo "this is run as joe")
```

Note: This command invokes DevAssistant under another user and passes the whole section to it. This means some behaviour differences from e.g. `scl` command, where each command is run in current assistant. Most importantly, RES of this command is RES of all commands from given subsection.

Using Another Section

Runs a section specified by **command input** at this place.

`use`, `call` (these two do completely same, `call` is obsolete and will be removed in 0.9.0) This can be used to run:

- another section of this assistant (e.g. `use: self.run_foo`)

- section of superassistant (e.g. `use: super.run`) - searches all superassistants (parent of this, parent of the parent, etc.) and runs the first found section of given name

- section from snippet (e.g. `use: snippet_name.run_foo`)

- Input: a string with section name

- RES: RES of the last command in the given section

- LRES: LRES of the last command in the given section

- Example:

```
– use: self.run_foo
– use: super.run
– use: a_snippet.run_spam
```

Normalizing User Input

Replace "weird characters" in user input by underscores.

- Input: a string

- RES: a string with weird characters (e.g. brackets/braces, whitespace, etc) replaced by underscores

- LRES: True

- Example:

  ```
  - $dir~:
    - normalize: foo!@#$%^bar
  - cl: mkdir $dir  # creates dir named foo_____bar
  ```

Setting up Project Directory

Creates a project directory (possibly with a directory containing it) and sets some global variables.

- Input: a mapping of input options, see below

- RES: path of project directory or a directory containing it, if `create_topdir` is `False`

- LRES: always True, terminates DevAssistant if something goes wrong

- Example:

  ```
  - $dir: foo/bar/baz
  - setup_project_dir:
      from: $dir
      create_topdir: normalized
  ```

Note: as a side effect, this command runner sets 3 global variables for you (their names can be altered by using arguments `contdir_var`, `topdir_var` and `topdir_normalized_var`):

- `contdir` - the dir containing project directory (e.g. `foo/bar` in the example above)

- `topdir` - the project directory (e.g. `baz` in the example above)

- `topdir_normalized` - normalized name (by *Normalizing User Input*) of the project directory

Arguments:

- `from` (required) - a string or a variable containing string with directory name (possibly a path)

- `create_topdir` - one of `True` (default), `False`, `normalized` - if `False`, only creates the directory containing the project, not the project directory itself (e.g. it would create only `foo/bar` in example above, but not the `baz` directory); if `True`, it also creates the project directory itself; if `normalized`, it creates the project directory itself, but runs it's name through *Normalizing User Input* first

- `contdir_var`, `topdir_var`, `topdir_normalized_var` - names to which the global variables should be assigned to - *note: you have to use variable names without dollar sign here*

- `accept_path` - either `True` (default) or `False` - if `False`, this will terminate DevAssistant if a path is provided

- `on_existing` - one of `fail` (default), `pass` - if `fail`, this will terminate DevAssistant if directory specified by `from` already exists; if `pass`, nothing will happen

1.2.7 Common Assistant Behaviour

Common Parameters of Assistants and Their Meanings

–e Create Eclipse project, optional. Should create `.project` (or any other appropriate file) and register project to Eclipse workspace (`~/workspace` by default, or the given path if any).

–g Register project on GitHub (uses current user name by default, or given name if any).

–n Name of the project to create, mandatory. Should also be able to accept full or relative path.

To include these parameters in your assistant with common help strings etc., include them from `common_args.yaml` (`-n`, `-g`) or `eclipse.yaml` (`-e`) snippet:

```
args:
  name:
    snippet: common_args
```

Other Conventions

When creating snippets/Python commands, they should operate under the assumption that current working directory is the project directory (not one dir up or anywhere else). It is the duty of assistant to switch to that directory. The benefit of this approach is that you just *cd* once in assistant and then call all the snippets/commands, otherwise you'd have to put 2x'cd' in every snippet/command.

1.2.8 Snippets

Snippets are the DevAssistant's way of sharing common pieces of assistant code. For example, if you have two assistants that need to log identical messages, you want the messages to be in one place, so that you don't need to change them twice when a change is needed.

Example

Let's assume we have two assistants like this:

```
### assistants/crt/assistant1.yaml
...
run:
- do: some stuff
- log_i: Creating cool project $name ...
- log_i: Still creating ...
- log_i: I suggest you go have a coffee ...
- do: more stuff

### assistants/crt/assistant2.yaml
...
run:
- do: some slightly different stuff
- log_i: Creating cool project $name ...
- log_i: Still creating ...
- log_i: I suggest you go have a coffee ...
- do: more slightly different stuff
```

So we have two assistants that have three lines of identical code in them - that breaks a widely known programmer best practice: Don't do it twice, write a function for it. In DevAssistant terms, we'll write a run section and place it in a snippet:

```
### snippets/mysnip.yaml
run:
- log_i: Creating cool project $name ...
- log_i: Still creating ...
- log_i: I suggest you go have a coffee ...
```

Then we'll change the two assistants like this (we'll utilize *"use" command runner*):

```
### assistants/crt/assistant1.yaml
...
run:
- do: some stuff
- use: mysnip.run
- do: more stuff


### assistants/crt/assistant2.yaml
...
run:
- do: some slightly different stuff
- use: mysnip.run
- do: more slightly different stuff
```

How Snippets Work

This section summarizes important notes about how snippets are formed and how they work.

Syntax and Sections

Snippets are very much like assistants. They can (but don't have to) have *args*, *dependencies** and *run** sections - structured in the same manner as in assistants. A snippet can contain any combination of the above sections (even empty file is a valid snippet).

Variables

When a snippet section is called (this applies to both *dependencies** and *run**, it gets a copy of all arguments of its caller - e.g. it can use the variables, it can assign to them, but they'll be unchanged in the calling section after the snippet finishes.

Using Snippets and Return Value

As noted above, snippets can hold 3 types of content (*args*, *dependencies** sections and *run** sections), each of which can be used in assistants:

```
### snippets/mysnip.yaml

args:
  foo:
    flags: [-f, --foo]
    help: Foo is foo
```

```
    required: True

dependencies:
- rpm: [python3]

run:
- log_i: Spam spam spam

### assistants/crt/assistant1.yaml

args:
  foo:
    use: mysnip

dependencies:
- use: mysnip.dependencies

run:
- do: stuff
- use: mysnip.run
```

Return values (RES and LRES) of snippet are determined by the *use command runner* - RES and LRES of last command of the snippet section.

1.2.9 Project Metainfo: the .devassistant File

Note: .devassistant file changed some of its contents and semantics in version 0.9.0.

Project created by DevAssistant usually get a `.devassistant` file, see *.devassistant Commands* for information on creating and manipulating it by assistants. This file contains information about a project, such as project type or paramaters used when this project was created. It can look like this:

```
devassistant_version: 0.9.0
original_kwargs:
  name: foo
  github: bkabrda
project_type: [python, django]
dependencies:
- rpm: [python-django]
```

When .devassistant is used

Generally, there are two use cases for `.devassistant`:

- Modifier assistants read the `.devassistant` file to get project type (which is specified by `project_type` entry) and decide what to do with this type of project (by choosing a proper `run` section to execute and proper `dependencies` section, see *Modifier Assistants*).

- When you use the `custom` preparer with URL to this project (`da prepare custom -u <url>`), DevAssistant will checkout the project, read the data from `.devassistant` and do few things:

    - It will install any dependendencies that it finds in `.devassistant`. These dependencies look like normal *dependencies section* in assistant, e.g.:

        ```
        dependencies:
        - rpm: [python-spam]
        ```

- It will also run a `run` section from `.devassistant`, if it is there. Again, this is a normal *run section*:

  ```
  run:
  - log_i: Hey, I'm running from .devassistant after checkout!
  ```

Generally, when using `custom` assistant, you have to be **extra careful**, since someone could put `rm -rf ~` or similar evil command in the `run` section. So use it **only with projects whose upstream you trust**.

1.2.10 Project Types

This is a list of official project types that projects should use in their `.devassistant` file and Creator assistants should state. If you choose one of the official project types, there is a good chance that Modifier and Preparer assistants written by others will work well with projects created by your Creator.

The project type is given as a list of strings - these describe the project from the most general type to the most specific. E.g:

```
project_type: [python, django]
```

If you don't use `project_type` in your Creator assistant, it will be automatically supported to `.devassistant`: If your assistant is `crt/footest/foobar.yaml`, project type in `.devassistant` will be `[footest, foobar]`. This means that Modifier and Preparer assistants written by others may not work well with your project, but otherwise it does no harm.

Current List of Types

Current project types list follows. If you want anything added in here, open a bug for us at https://github.com/bkabrda/devassistant/issues. **Note: the list is currently not very thorough and it is meant to grow as we get requested by assistant developers.**

- c
- cpp
- java
- nodejs
    - express
- perl
    - dancer
- php
- python
    - django
    - flask
    - gtk3
    - lib
- ruby
    - rails

1.2.11 Talk to Us!

If you want to see where DevAssistant development is going and you want to influence it and send your suggestions and comments, you should *really* join our ML: https://lists.fedoraproject.org/mailman/listinfo/devassistant.
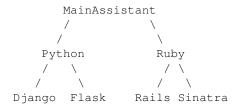
1.2.12 Overall Design

DevAssistant consists of several parts:

Core Core of DevAssistant is written in Python. It is responsible for interpreting Yaml Assistants and it provides an API that can be used by any consumer for the interpretation.

CL Interface CL interface allows users to interact with DevAssistant on commandline; it consumes the Core API.

GUI (work in progress) GUI allows users to interact with Developer Assistant from GTK based GUI; it consumes the Core API.

Assistants Assistants are Yaml files with special syntax and semantics (defined in *Yaml Assistant Reference*). They are indepent of the Core, therefore any software distribution can carry its own assistants and drop them into the directory from where DevAssistant loads them - they will be loaded on next invocation. Note, that there is also a possibility to write assistants in Python, but this is no longer supported and will be removed in near future.

1.2.13 Assistants

Internally, each assistant is represented by instance of `devassistant.yaml_assistant.YamlAssistant`. Instances are constructed by DevAssistant in runtime from parsed yaml files. Each assistant can have zero or more subassistants. This effectively forms a tree-like structure. For example:

```
      MainAssistant
      /           \
     /             \
  Python          Ruby
  /  \            /  \
 /    \          /    \
Django  Flask   Rails Sinatra
```

This structure is defined by filesystem hierarchy as explained in *Assistants Loading Mechanism*

Each assistant can optionally define arguments that it accepts (either on commandline, or from GUI). For example, you can run the leftmost path with:

```
$ da create python [python assistant arguments] django [django assistant arguments]
```

If an assistant has any subassistants, one of them **must** be used. E.g. in the example above, you can't use just Python assistant, you have to choose between Django and Flask. If Django would get a subassistant, it wouldn't be usable on its own any more, etc.

Assistant Roles

The `crt` in the above example means, that we're running an assistant that creates a project.

There are four assistant roles:

creator (`create` or `crt` on command line) creates new projects

modifier (`modify` or `mod` on command line) works with existing projects

preparer (`prepare` or `prep` on command line) prepares environment for development of upstream projects

task (`task` on command line) performs arbitrary tasks not related to a specific project

The main purpose of having roles is separating different types of tasks. It would be confusing to have e.g. `python django` assistant (that creates new project) side-by-side with `eclipse` assistant (that registers existing project into Eclipse).

1.2.14 Contributing

If you want to contribute (bug reporting, new assistants, patches for core, improving documentation, ...), please use our Github repo:

- code: https://github.com/bkabrda/devassistant

- issue tracker: https://github.com/bkabrda/devassistant/issues

If you have DevAssistant installed (version 0.8.0 or newer), there is a fair chance that you have `devassistant` preparer. Just run `da prepare devassistant` and it will checkout our sources and do all the boring stuff that you'd have to do without DevAssistant.

If you don't have DevAssistant installed, you can checkout the sources like this (just copy&paste this to get the job done):

```
git clone https://github.com/bkabrda/devassistant
# get the official set of assistants
cd devassistant
git submodule init
git submodule update
```

You can find list of core Python dependencies in file `requirements.txt`. If you want to write and run tests (you should!), install dependencies from `requirements-devel.txt`:

```
pip install -r requirements-devel.txt
```

On top of that, you'll need `polkit` for requesting root privileges for dependency installation etc. If you want to play around with GUI, you have to install `pygobject`, too (see how hard this is compared to `da prepare devassistant`?)

Overview

This is documentation for version 0.9.2.

DevAssistant is developer's best friend (right after coffee).

DevAssistant (http://devassistant.org) can help you with creating and setting up basic projects in various languages, installing dependencies, setting up environment etc. There are four main types of functionality provided:

- `da create` - create new project from scratch
- `da modify` - take local project and do something with it (e.g. import it to Eclipse)
- `da prepare` - prepare development environment for an upstream project or a custom task
- `da task` - perform a custom task not related to a specific project

The first three of these have shortcuts for faster use: "create" can be shortened as "crt", "modify" as "mod" and "prepare" as "prep".

DevAssistant is based on idea of per-{language/framework/...} "assistants" with hierarchical structure. E.g. you can run:

```
$ da create python django -n ~/myproject # sets up Django project named "myproject" inside your home
$ da create python flask -n ~/flaskproject # sets up Flask project named "flaskproject" inside your h
$ da create ruby rails -n ~/alsomyproject # sets up RoR project named "alsomyproject" inside your hom
```

DevAssistant also allows you to work with a previously created project, for example import it to Eclipse:

```
$ da modify eclipse # run in project dir or use -p to specify path
```

With DevAssistant, you can also prepare environment for developing upstream projects - either using project-specific assistants or using "custom" assistant for arbitrary projects (even those not created by DevAssistant):

```
$ da prepare custom custom -u scm_url
```

Warning: The `custom` assistant executes custom pieces of code from `.devassistant` file of the project. Therefore you have to be extra careful and use this **only with projects whose authors you trust**.

Last but not least, DevAssistant allows you to perform arbitrary tasks not related to a specific project:

```
$ da task <TODO:NOTHING YET>
```

Should you have some questions, feel free to ask us at Freenode channel #devassistant or on our mailing list (https://lists.fedoraproject.org/mailman/listinfo/devassistant). You can also join our G+ community (https://plus.google.com/u/0/communities/112692240128429771916) or follow us on Twitter (https://twitter.com/dev_assistant).

DevAssistant works on Python 2.6, 2.7 and >= 3.3.

This whole project is licensed under GPLv2+.