# A practical guide to BitBake

Harald Achitz – harald.achitz@gmail.com – Version 1.3, 2019 Q1

## Table of Contents

# 1. Preface

## 1.1. About this tutorial

If you read this tutorial you probably know already that BitBake is used as a build tool, primary by the OpenEmbedded and the Yocto project, to build Linux distributions. You might also have noticed that working with BitBake has a somewhat steep learning curve. This document was made to flatten this curve.

This document does not tell you everything about BitBake, this is in short not possible, but it tries to

explain some of the fundamental functionalities that BitBake uses. Understanding these basics should help if you ever start to write your own BitBake recipes.

### 1.2. Target of this tutorial

The tutorial shows how to create the smallest possible project and extend it step by step to show and explain how BitBake works

### 1.3. Acknowledgments

Thanks to Tritech for giving me some time to prepare the basic foundation for this document back in 2014. Since then, the this tutorial has evolved, but without the very first version this document would not exist.

A lot of thanks to the people that report issues and typos to the issue tracker for this site!

### 1.4. Feedback

If you find bugs, unclear sections, typos or simply have suggestions please use the issue tracker at:
https://bitbucket.org/a4z/bitbakeguide/issues
No registration required.

Also feel free to use the Disqus comment function at the end of the document.

## 2. BitBake

### 2.1. What is BitBake

When working with BitBake it helps to understand the following:
Basically BitBake is a Python program which, driven by user created configuration, can execute user created tasks for user specified targets, so called recipes.

### 2.1.1. Config, tasks and recipes

Configuration, tasks and recipes are written in a kind of BitBake DSL (Domain Specific Language) which contain variables and executable shell or python code. So in theory, since BitBake executes code, someone could use BitBake for something else than building software, but this would possibly be not the best idea.

BitBake was made as a tool to build software and has therefore some special features, for example the possibility to define dependencies. BitBake is able to resolve dependencies and put the work it has to do into the right order. Furthermore, building software packages contains often equal or very similar task. Common tasks are for example: download and extract the source code, run configure, run make, or simply write a log message. BitBake provides a mechanism to abstract, encapsulate and reuse this functionality in a configurable way.

## 3. Setup BitBake

Bitbake can be downloaded here:
https://github.com/openembedded/bitbake
Select a branch with a version and download the zip. Extract the zip archive into some folder, it will place a bitbake-$version folder there.

This tutorial has been updated to work with with **python 2.7.6** and **bitbake 1.40.0**. All the examples have been adopted to work with these versions, what is actual state at begin of 2019.

This tutorial was initially written several years ago. Since then, both, python and bitbake made incompatible changes and this might happen again in future. If you find any problems, please report them in the comment section, or via the issue tracker. In case of problems, to continue working with this tutorial, ensure you use bitbake and python in the versions as mentioned above.

If used within Yocto there is no requirement to install BitBake, it will come bundled with the Yocto sources. Yocto will require you to source a script, and this script does the same as we do now, installing BitBake in our environment.

## 3.1. The installation of BitBake

The installation is very simple:

- Add bitbake-$version/bin directory to PATH
- Add bitbake-$version/lib directory to PYTHONPATH

We can do this by running

```
export PATH=/path/to/bbtutor/bitbake/bin:$PATH
export PYTHONPATH=/path/to/bbtutor/bitbake/lib:$PYTHONPATH
```

This is basically the same as the yocto init script does.
The yocto init script does also creates a build folder, we will do that later.

First we check if everything works and bitbake is installed.
To do that run the following bitbake command:

```
bitbake --version
```

It should print something like:

```
BitBake Build Tool Core version 1.22.0, bitbake version 1.22.0
```

## 3.2. The BitBake documentation

The most actual version comes with the source code.

In a terminal, cd into the bitake-$version/doc directory and run

```
make html DOC=bitbake-user-manual
```

to create doc/bitbake-user-manual/bitbake-user-manual.html.

This document can be read in parallel to this tutorial and needs to be read after reading this tutorial.

The yocto project documentation has a bitbake section for the version it uses in the release.

# 4. Create a project

## 4.1. Bitbake project layout

Usually a BitBake project is organized in folders with configuration and meta data, called layers, and a build folder.

### 4.1.1. Layer folder

A layer folder contains configuration, task and target descriptions for what BitBake does.
It is common practice to name a layer folders meta-'something'.

### 4.1.2. Build folder

The build folder is the directory from which the `bitbake` command has to be executed. Here, BitBake expects its initial configuration file and it will place all files it creates in this folder.

To be able to run BitBake without getting any errors we need to create a build and a layer folder and place some required configuration files in there.

## 4.2. The smallest possible project

The minimal configuration will look like this:

```
bbTutorial/
├── build
│   ├── bitbake.lock
│   └── conf
│       └── bblayers.conf
└── meta-tutorial
    ├── classes
    │   └── base.bbclass
    └── conf
        ├── bitbake.conf
        └── layer.conf
```

These 4 files

- bblayers.conf
- base.bbclass
- bitbake.conf
- layer.conf

need to be created next.

### 4.2.1. The required config files

First a description of the needed files, then a short description to the content.

#### build/conf/bblayers.conf

The first file BitBake expects is conf/bblayers.conf in its working directory, which is our build directory.
For now we create it with this content:

build/conf/bblayers.conf

```
BBPATH := "${TOPDIR}"
BBFILES ?= ""
BBLAYERS = "${TOPDIR}/../meta-tutorial"
```

#### meta-tutorial/conf/layer.conf

Each layer needs a conf/layer.conf file. For now we create it with this content:

meta-tutorial/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += ""
```

#### meta-tutorial/classes/base.bbclass

#### meta-tutorial/conf/bitbake.conf

For now, these files can be taken from the BitBake installation directory.
These files are located in the folders *bitbake-$version/conf* and *bitbake$version/classes*.
Simply copy them into the tutorial project.

### 4.2.2. Some notes to the created files

#### build/conf/bblayers.conf

- Add the current directory to BBPATH.
  TOPDIR is internally set by BitBake to the current working directory.

- Initialize the variable BBFILES as empty. Recipes will be added later.
- Add the path of our meta-tutorial to the BBLAYERS variable.
  When executed, BitBake will search all given layer directories for additional configurations.

**meta-tutorial/conf/layer.conf**

- LAYERDIR is a variable BitBake passes to the layer it loads.
  We append this path to the BBPATH variable.
- BBFILES tells BitBake where recipes are.
  Currently we append nothing, but we will change this later.

The ".=" and "+=" append the value without or with space to a variable.
This, and other notation is documented BitBake documentation chapter 3.1..*

**conf/bitbake.conf**

The conf/bitbake.conf contains a bunch of variables which, for now, we just take as they are.

**classes/base.bbclass**

A *.bbclass file contains shared functionality.
Our base.bbclass contains some logging functions which we will use later, and a build task that does nothing.
Not very useful, but required by BitBake since `build` is the task BitBake runs per default if no other task is specified. And we will change this function later.

The BitBake manual section 3.3 describes Sharing Functionality.

### 4.2.3. BitBake search path

For BitBake there are some file paths which are relative to BBPATH.

This means that if we tell BitBake to search for some path then it will search all directives in BBPATH for somedir/somefile.
We have added TOPDIR and LAYERDIR to BBPATH, so classes/base.bbclass and conf/bitbake.conf could be in any of them.
But of course we added them to the meta-tutorial directory.
The build directory should never contain general files, only special files like a local.conf which is valid just for the actual build. We will use a local.conf later.

## 4.3. The first run

In a terminal change into the just created *build* directory, which is our working directory.
We always run `bitbake` from the build directory so that `bitbake` can find the relative *conf/bblayers.conf* file.

Now simply run the `bitbake` without any arguments.

[~/bbTutorial/build]

```
bitbake
```

If our setup is correct `bitbake` will report:

```
Nothing to do.  Use 'bitbake world' to build everything,
or run 'bitbake --help' or usage information.
```

This is not very useful at all, but a good start.

And this is a good opportunity to introduce a nice and useful command flag, which is: verbose some debug output.

To see it in action run

[~/bbTutorial/build]

```
bitbake -vDDD world
```

and check the output, It tells us already a lot about how BitBake works.

The output you will see might look similar this one:

```
DEBUG: Found bblayers.conf (~/bbguilde/bitbakeguide/ch04/build/conf/bblayers.conf)
DEBUG: Adding layer ~/bbguilde/bitbakeguide/ch04/build/../meta-tutorial
DEBUG: Inheriting ~/bbguilde/bitbakeguide/ch04/build/../meta-tutorial/classes/base.bbclass (from
configuration INHERITs:0)
DEBUG: Clearing SRCREV cache due to cache policy of: clear
DEBUG: Using cache in '~/bbguilde/bitbakeguide/ch04/build/tmp/cache/local_file_checksum_cache.dat'
DEBUG: Using cache in '~/bbguilde/bitbakeguide/ch04/build/tmp/cache/bb_codeparser.dat'
DEBUG: Features set [2] (was [2])
DEBUG: Updating new environment variable LC_ALL to en_US.UTF-8
DEBUG: Base environment change, triggering reparse
DEBUG: Found bblayers.conf (~/bbguilde/bitbakeguide/ch04/build/conf/bblayers.conf)
DEBUG: Adding layer ~/bbguilde/bitbakeguide/ch04/build/../meta-tutorial
DEBUG: Inheriting ~/bbguilde/bitbakeguide/ch04/build/../meta-tutorial/classes/base.bbclass (from
configuration INHERITs:0)
DEBUG: Clearing SRCREV cache due to cache policy of: clear
DEBUG: Using cache in '~/bbguilde/bitbakeguide/ch04/build/tmp/cache/local_file_checksum_cache.dat'
DEBUG: Using cache in '~/bbguilde/bitbakeguide/ch04/build/tmp/cache/bb_codeparser.dat'
DEBUG: collecting .bb files
ERROR: no recipe files to build, check your BBPATH and BBFILES?
NOTE: Not using a cache. Set CACHE = <directory> to enable.

Summary: There was 1 ERROR message shown, returning a non-zero exit code.
```

The argument ` -vDDD` tells bitbake to be as verbose as possible, and `world` is a target to build. We will learn more about targets later.

Since we have no recipe and target world, bitbake exits with an error. This is OK for now and we will fix this in the next chapter.

Did you notice that BitBake create a tmp directory?

All example code is available at https://bitbucket.org/a4z/bitbakeguide
The example for this chapter is ch04.

## 5. The first recipe

BitBake needs recipes to do something. Currently there is none, so even if we run the `bitbake` command, without having a recipe it makes it not that much fun.

We can easily verify that there is nothing to do by running

[~/bbTutorial/build]

```
bitbake -s
```

Running this command will report:

```
NOTE: Not using a cache. Set CACHE = <directory> to enable.
Recipe Name    Latest Version    Preferred Version
===========    ==============    =================
```

This tells us 2 things:

1. BitBake tells us that it has no cache defined.
2. BitBake tells us that it has really nothing to do by showing us an empty list

## 5.1. The cache location

BitBake caches meta data information in a directory, the cache. This help to speed up subsequent execution of commands.

We can fix the missing cache by simply adding a variable to bitbake.conf.
Therefore we edit the *meta-tutorial/conf/bitbake.conf* file and add at the end:

meta-tutorial/conf/bitbake.conf

```
...
CACHE = "${TMPDIR}/cache/default"
```

This is OK for now.

In real projects, like Yocto, this variable is already set and we do not need to care about it.
Often the cache path is composed out of different variables to have the actual build configuration, like debug or release, in the name.

The next step is to add a recipe which requires 2 steps:

1. enable bitbake to find recipes
2. write a first recipe

## 5.2. Adding a recipe location to the tutorial layer

BitBake needs to know about which recipes a layer provides.
We edit our *meta-tutorial/conf/layer.conf* file and tell BitBake to load all recipe files by using a common pattern.

meta-tutorial/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
```

We make now use of the variable previously defined in *build/conf/bblayers.conf*. A recipe file has the extension *.bb, and if we respect the common pattern we can simply add all recipes to BitBake with one line.

Usually recipes have their own folder and are collected in groups, which means put recipes that are somehow related into the same directory.

It is common to name these folders recipes-'group', where group names a category of programs.

Now, since BitBake knows where to find recipes, we can actually add our fist one.

Following the common pattern we create the folders *meta-tutorial/recipes-tutorial/first* and create the first recipe in there. Recipe-files also have a common name pattern which is `{recipe}_{version}.bb`.

## 5.3. Create the first recipe and task

Our first recipe will just print a log message. We put it into the first group, we will call it first, and it has the version 0.1.

So our **first** recipe in the first group is:

meta-tutorial/recipes-tutorial/first/first_0.1.bb

```
DESCRIPTION = "I am the first recipe"
```

```
PR = "r1"
do_build () {
  echo "first: some shell script running as build"
}
```

- The **task** do_build overrides the empty global build task from base.bbclass.
- **PR** is the internal revision number which should be updated after each change.
- Setting a **description** should be self explaining.

If everything is done correct we can ask bitbake to list the available recipes.

[~/bbTutorial/build]

```
bitbake  -s
Parsing recipes: 100% ...
Parsing of 1 .bb files complete...
Recipe Name    Latest Version   Preferred Version
===========    ==============   =================
first                 :0.1-r1
```

and we can run from the build directory

[~/bbTutorial/build]

```
bitbake first
```

Now check *tmp/work/first-0.1-r1/temp*, there are a lot of interesting files, for example:

build/tmp/work/first-0.1-r1/temp/log.do_build

```
DEBUG: Executing shell function do_build
```

```
first: some shell script running as build
DEBUG: Shell function do_build finished
```

All example code is available at https://bitbucket.org/a4z/bitbakeguide
The example for this chapter is ch05.

# 6. Classes and functions

The next steps will be:

- Add a class
- Add a recipe that uses this class.
- Explore functions

## 6.1. Create the mybuild class

Let's create a different build function and share it.
We can do this by creating a class in the tutorial layer.
Therefore we create a new file called *meta-tutorial/classes/mybuild.bbclass*.

meta-tutorial/classes/mybuild.bbclass

```
addtask build
mybuild_do_build () {

  echo "running mybuild_do_build."

}

EXPORT_FUNCTIONS do_build
```

As in base.class, we add a build task. It is again a simple shell function.
*mybuild_do_* prefix is for following the conventions, *classname_do_functionname* for a task in a class.

EXPORT_FUNCTIONS makes the build function available to users of this class.
If we did not have this line it could not override the build function from base for.

For now this is enough to use this class with our second recipe.

## 6.2. Use myclass with the second recipe

Time to build a second recipe which shall use the build task defined in mybuild.
Say that this target needs to run a patch function before the build task.
And as an additional challenge second shall also demonstrate some python usage.

Following bitbakes naming conventions we add a new recipe folder and add the file *meta-tutorial/recipes-tutorial/second/second_1.0.bb* to it.

The new file wll look like this.

meta-tutorial/recipes-tutorial/second/second_1.0.bb

```
DESCRIPTION = "I am the second recipe"
PR = "r1"                          1
inherit mybuild                    2

def pyfunc(o):                     3
    print(dir(o))

python do_mypatch () {             4
  bb.note ("runnin mypatch")
  pyfunc(d)                        5
}

addtask mypatch before do_build    6
```

1. DESCRIPTION and PR are as usual.
2. The mybuild class becomes inherited and so myclass_do_build becomes the default build task.
3. The (pure python) function pyfunc takes some argument and runs the python dir function on this argumentm and prints the result.
4. The (bitbake python) mypatch function is added and registered as a task that needs to be executed before the build function.
5. mypatch calls pyfunc and passes the global bitbake variable d.
   d (datastore) is defined by bitbake and is always available.
6. The mypatch function is registered as a task that needs to be executed before the build function.

Now we have an example that uses python functions.

The functions part in the bitbake manual is section 3.4.

## 6.3. Exploring recipes and tasks

Having now two recipes we are able to use and explore additional `bitbake` command options.

We can get information about recipes and their tasks and control what BitBake will execute.

### 6.3.1. List recipes and tasks

First we can check if BitBake really has both recipes. We can do this by using the *-s* option.

[~/bbTutorial/build]

```
bitbake -s
```

will now output

```
Recipe Name    Latest Version  Preferred Version
===========    ==============  =================
first                 :0.1-r1
second                :1.0-r1
```

If we would like to see all tasks a recipe provides we can explore them with `bitbake -c listtasks second`

This should give us a first overview on how to explore recipes and tasks.

## 6.4. Executing tasks or building the world

We have now several options on running builds or specified tasks for our recipes.

**Build one recipe**

To run all tasks for our second recipe we simply call `bitbake second`

**Execute one task**

We could also run a specific task for a recipe.
Say we want only to run the mypatch task for the second recipe.
This can be done by applying the command `bitbake -c mypatch second`

**Build everything**

Simply running all tasks for all recipes can be done with `bitbake world`

You can play with the commands and see what happens.
The console output will tell you what was executed.

### 6.4.1. Checking the build logs

Bitbake creates a tmp/work directory in its actual build location where it stores all log files. These log files contain interesting information and are worth to study.

An actual output after a first `bitbake world` run might look like this.

```
tmp/work/
  |- first-0.1-r1
    |- temp
      |-log.do_build -> log.do_build.20703
      |-log.do_build.20703
      |-log.task_order
      |-run.do_build -> run.do_build.20703
      |-run.do_build.20703
  |- second-1.0-r1
    |- second-1.0
    |- temp
      |-log.do_build -> log.do_build.20706
      |-log.do_build.20706
      |-log.do_mypatch -> log.do_mypatch.20705
      |-log.do_mypatch.20705
      |-log.task_order
      |-run.do_build -> run.do_build.20706
      |-run.do_build.20706
      |-run.do_mypatch -> run.do_mypatch.20705
      |-run.do_mypatch.20705
```

These log files contain useful information from BitBake about its actions as well as the output of the executed tasks.

All example code is available at https://bitbucket.org/a4z/bitbakeguide
The example for this chapter is ch06.

# 7. BitBake layers

A typical BitBake project consists of more than one layer.

Usually layers contain recipes to a specific topic. Like basic system, graphical system, ...and so on.

In some project there might also be more than one build target and each target is composed out of different layers.
A typical example would be to build a Linux distribution with and without GUI components.

Layers can used, extend, configured and it is also possible to partial overwrite parts of existing layers.
This is useful since it allows reuse and customization for actual needs.

Working with multiple layers is the common case and therefore we also add an additional layer to the project.

## 7.1. Adding an additional layer

Adding a new layer can be done with the following steps:

1. Create the new layer folder
2. Create the layer configuration
3. Tell BitBake about the new layer
4. Add recipes to the layer

### 7.1.1. Adding the new layer folder

Create a new folder named *meta-two*.
We follow the common naming conventions and our working place looks now like this:

```
ls ~/bbTutorial
build  meta-tutorial  meta-two
```

### 7.1.2. Configure the new layer

Add *meta-two/config/layer.conf* file. This file looks exactly like the one for the tutorial layer.

meta-two/config/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
```

### 7.1.3. Telling the BitBake about the new layer recipes

edit build/conf/bblayers.conf and extend the BBLAYERS variable.

build/conf/bblayers.conf

```
BBLAYERS = " \
  ${TOPDIR}/../meta-tutorial \
  ${TOPDIR}/../meta-two \
"
```

## 7.2. The bitbake-layer command

The tool to explore layer configurations is the `bitbake-layer` command.

The `bitbake-layers` command has a variety of useful options, the help text says it all.

```
Available commands:
  help
    display general help or help on a specified command
```

```
show-recipes
  list available recipes, showing the layer they are provided by
show-cross-depends
  figure out the dependency between recipes that crosses a layer boundary.
show-appends
  list bbappend files and recipe files they apply to
flatten
  flattens layer configuration into a separate output directory.
show-layers
  show current configured layers
show-overlayed
  list overlayed recipes (where the same recipe exists in another layer)
```

To explore, for example, which layers exists, you can run

[~/bbTutorial/build]

```
bitbake-layers show-layers
```

But before this will produce any useful output for our project we need to adopt our layers configurations.

## 7.3. Extending the layer configuration

We need to add some information to our layer configuration.

- a layer collection name
- a search pattern for files to add
- a layer priority

We start with *meta-tutorial/conf/layer.conf* and add

meta-tutorial/conf/layer.conf

```
# append layer name to list of configured layers
BBFILE_COLLECTIONS += "tutorial"
# and use name as suffix for other properties
BBFILE_PATTERN_tutorial = "^${LAYERDIR}/"
BBFILE_PRIORITY_tutorial = "5"
```

The used variables have an excellent description in the BitBake user manual, so there is no need to repeat this text here.

The patterns should be clear, we define the layer name and use this name to suffix some other variables.
This mechanism, using user defined domain suffixes in BitBake variable names, is used by BitBake on several locations.

Next we change meta-two/conf/layer.conf in the same way.

meta-two/conf/layer.conf

```
BBFILE_COLLECTIONS += "two"
BBFILE_PATTERN_two = "^${LAYERDIR}/"
BBFILE_PRIORITY_two = "5"
LAYERVERSION_two = "1"
```

If we now run `bitbake-layers show-layers` it will report

```
layer        path                                priority
==========================================================
meta-tutorial   /path/to/work/build/../meta-tutorial  5
meta-two     /path/to/work/build/../meta-two    5
```

All example code is available at https://bitbucket.org/a4z/bitbakeguide
The example for this chapter is ch07.

## 7.4. Layer compatibility

A project like yocto is composed out of very many layers. To ensure used layer are compatible with a project version, a project can define a *layer series* name, and layers can specify to be compatible to one, or multiple, *layer series*.

In practice, for a yocto project, each release defines its release name as its *layer series core name*. Layers that are tested for this release can add the compatibility name in its config. If a layer is added that does not have the compatibility name specified, bitbake will tell about this by showing a warning.

We can easily verify this. So far there is no core layer series name specified in our tutorial. Running, for example, `bitbake-layers show-recipes` will give 5 warnings.

```
 bitbake-layers show-recipes
NOTE: Starting bitbake server...
WARNING: Layer tutorial should set LAYERSERIES_COMPAT_tutorial in its conf/layer.conf file to list
the core layer names it is compatible with.
WARNING: Layer two should set LAYERSERIES_COMPAT_two in its conf/layer.conf file to list the core
layer names it is compatible with.
WARNING: Layer tutorial should set LAYERSERIES_COMPAT_tutorial in its conf/layer.conf file to list
the core layer names it is compatible with.
WARNING: Layer two should set LAYERSERIES_COMPAT_two in its conf/layer.conf file to list the core
layer names it is compatible with.
Parsing recipes: 100%
|##################################################################################################
##########################| Time: 0:00:00
Parsing of 2 .bb files complete (0 cached, 2 parsed). 2 targets, 0 skipped, 0 masked, 0 errors.
WARNING: No bb files matched BBFILE_PATTERN_two '^/home/bitbakeguide/ch07/build/../meta-two/'
```

```
Summary: There were 5 WARNING messages shown.
=== Available recipes: ===
first:
  meta-tutorial        0.1
second:
  meta-tutorial        1.0
```

The first 4 warning referee to the fact that the tutorial project has no layer series compatibility specified. The fifths warning is because layer two is empty, what will fix this in the next chapter. First, lets add a layer series name and specify that the 2 layers in the tutorial project are compatible.

### 7.4.1. Layer series core name

First we define a 'project core name'. This is done by setting the `LAYERSERIES_CORENAMES` variable. In yocto, this is done in the core layer, a layer with the name *core*.
We define the name in the tutorial layer because it is our first layer. The actual location does not matter, it could also be defined in the build/conf/bblayers.conf file.

We set the core name by adding

```
LAYERSERIES_CORENAMES = "bitbakeguilde"
```

to .meta-tutorial/conf/layer.conf

### 7.4.2. Layer series compatibility

We also need to specify that the tutorial layer is compatible with the `bitbakeguilde`.
This can be done by setting `LAYERSERIES_COMPAT_...` variable in the in the layer.conf files of each layer. The variable ends on the layer name like we have seen it with the BBFILE_PATTERN or the BBFILE_PRIORITY variable.

Our layer configuration files looks now like that:

meta-tutorial/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
BBFILE_COLLECTIONS += "tutorial"
BBFILE_PATTERN_tutorial = "^${LAYERDIR}/"
BBFILE_PRIORITY_tutorial = "5"

LAYERSERIES_CORENAMES = "bitbakeguilde"

LAYERVERSION_tutorial = "1"
LAYERSERIES_COMPAT_tutorial = "bitbakeguilde"
```

meta-two/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "two"
BBFILE_PATTERN_two = "^${LAYERDIR}/"
BBFILE_PRIORITY_two = "5"
LAYERVERSION_two = "1"
LAYERDEPENDS_two = "tutorial"

LAYERSERIES_COMPAT_two = "bitbakeguilde"
```

With these changes the four warnings about missing compatibility information are gone. All our layers are declared compatible to the core layer series.

## 7.5. Layer dependencies

You might have noticed that we also specified the `LAYERDEPENDS_two` variable in the layer.conf file of our second layer, meta-two.

By doing so we inform bitbake that this layer has a dependency to the tutorial layer. We well see in the next chapter, when we add more content to the meta-two layer, why this is the case.

Play around, try our what happens if a compat name is not set, or wrong spelled, and run the `show-layers` command with different arguments.

# 8. Share and reuse configurations

So far we used classes and config files to encapsulate configuration and tasks.
But there are more ways to reuse and extend tasks and configurations.
These are:

- class inheritance
- bbappend files
- include files

To demonstrate these usages we are going to add an additional class to layer-two.
The new class will introduce a configure-build chain and will reuse the existing mybuild class by using class inheritance.
Then we will use this new class within a new recipe.
After that we will extend an existing recipe by using the append technique.

## 8.1. Class inheritance

To realize our configure/build chain we create a class that inherits the mybuild class and simply adds a configure task as a dependency of the build task.

We create this as another class, which we will then use to demonstrate a class and a recipe which make use of inheritance.

meta-two/classes/confbuild.bbclass

```
inherit mybuild                              1

confbuild_do_configure () {                  2

  echo "running configbuild_do_configure."

}

addtask do_configure before do_build         3

EXPORT_FUNCTIONS do_configure                4
```

1. use the mybuild class as a base
2. create the new function
3. define the order of the functions, configure before build
4. export the function so that it becomes available

We can now simply use this in our **third** recipe and use confbuild.

meta-two/recipes-base/third_01.bb

```
DESCRIPTION = "I am the third recipe"
PR = "r1"
inherit confbuild
```

This recipe inherits the confbuild class.

If we run now `bitbake third` it will execute the configure and build tasks for third.

## 8.2. bbappend files

An append file can be use to add functions to an existing class without creating a new one.
It adds the text of the append file to a class with the same name.

To be able to use append files the layer needs to be set up to load also them in addition to normal recipes.
Therefore we change our layer configuration and add loading of *.bbappend file to the BBFILES variable.

meta-two/conf/layer.conf

```
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"
```

Now we want to reuse and extend the existing first recipe.

This is why we added the `LAYERDEPENDS_two` in the previous chapter, we need this layer because it contains the recipe we want to extend.

What we want is running a patch function before running the build task, so we need to create the corresponding bbappend file and add our changes.
Therefore we need to create the *meta-two/recipes-base/first/* folder and the *first_0.1.bbappend* file.

meta-two/recipes-base/first/first_0.1.bbappend

```
python do_patch () {
  bb.note ("first:do_patch")
}

addtask patch before do_build
```

If we now list the tasks for the first recipe, we will see that it also has a patch task.

[~/bbTutorial/build]

```
bitbake -c listtasks first
Parsing recipes: 100% ...
....
do_showdata
do_build
do_listtasks
do_patch
...
```

Running `bitbake first` will now run both tasks, patch and build.

If you want, you can now build a recipe that uses the confbuild class and an append file to run patch, configure and build tasks.

## 8.3. Include files

BitBake has two directives to include files.

- `include` filename this is an optional include, if filename is not found no error is raised.
- `require` filename if filename is not found an error is raised.

It is worth mentioning that the include and require file name is relative to any directory in BBPATH.

### 8.3.1. Add a local.conf for inclusion

A common use case in BitBake projects is that the bitbake.conf includes a local.conf file which is usually placed in the build directory.
The local.conf file may contains special setups for the current build target.
This is the typical Yocto setup.

We mimic the typical usage of a local.conf and make bitbake.conf require a local.conf by adding the following lines to our `meta-tutorial/conf/bitbake.conf`.

meta-tutorial/conf/bitbake.conf

```
require local.conf
include conf/might_exist.conf
```

If we run now some build BitBake will show a long error message where the last line will be something like:

```
ERROR: Unable to parse conf/bitbake.conf: …Could not include required file local.conf
```

Adding a local.conf into the *build* directory, which can even be empty, will fix this.
The file given to the include statement is not required to exist.

All example code is available at https://bitbucket.org/a4z/bitbakeguide
The example for this chapter is ch08

## 9. Using Variables

The ability to define variables and use them in recipes makes BitBake very flexible.

Recipes can be written in a way that the configurable parts use variables.
The user of such recipes can give those variable values which then will be used by the recipes.
A typical example is passing extra configuration or make flags to a recipe.
By using variables properly there is no need to edit and change a recipe just because we need special arguments for some functions.

## 9.1. Global variables

Global variables can be set by the user and existing recipes can use them.

### 9.1.1. Define global variables

An empty local.conf, as we have currently, is not very useful. So let's add some variable to local.conf. Assuming we are in the build directory we can run:

[~/bbTutorial/build]

```
echo 'MYVAR="hello from MYVAR"' > local.conf
```

or use our favorite editor to add this to the file.

### 9.1.2. Accessing global variables

We can access `MYVAR` in recipes or classes. For demonstration we create the new recipe group recipes-vars and a recipe myvar in it.

meta-two/recipes-vars/myvar/myvar_0.1.bb

```
DESCRIPTION = "Show access to global MYVAR"
PR = "r1"

do_build(){
  echo "myvar_sh: ${MYVAR}"                          1
}

python do_myvar_py () {
  print ("myvar_py:" + d.getVar('MYVAR', True))       2
}

addtask myvar_py before do_build
```

1. Access the variable in a bash like syntax.
2. Access the variable via the global data store.

If we now run `bitbake myvar` and check the log output in the tmp directory, we will see that we indeed have access to the global MYVAR variable. If you are looking for the log file, search for a file like this: `build/tmp/work/myvar-0.1-r1/temp/log.do_myvar_py`.

## 9.2. Local variables

A typical recipe mostly consists only of variables that are used to set up functions defined in classes which the recipe inherits.

To have an idea how this work create

meta-two/classes/varbuild.bbclass

```
varbuild_do_build () {
  echo "build with args: ${BUILDARGS}"
}

addtask build

EXPORT_FUNCTIONS do_build
```

and use this class in a recipe

meta-two/recipes-vars/varbuild/varbuild_0.1.bb

```
DESCRIPTION = "Demonstrate variable usage \
  for setting up a class task"
PR = "r1"

BUILDARGS = "my build arguments"
```

```
inherit varbuild
```

Running `bitbake varbuild` will produce log files that shows that the build task respects the variable value which the recipe has set.

This is a very typical way of using BitBake. The general task is defined in a class, like for example download source, configure, make and others, and the recipe sets the needed variables for the task.

All example code is available at https://bitbucket.org/a4z/bitbakeguide
The example for this chapter is ch09

## 10. Summary

That's it with this tutorial, I am glad you made it until here and I hope you liked it.
After reading this tutorial you should have a solid foundation on the basics concepts BitBake is based on.

Topics covered are:

- BitBake as an engine that executes python and/or shell scripts.
- The common BitBake project layout and the default file locations.
- The basic understanding for layers and their relations to each other.
- The 5 file types BitBake uses (bb- bbclass- bbappend- conf- and include files).
- BitBake functions and tasks, show how to organize, group and call them.
- BitBake variables and the basic usage of them.

Being familiar with these topics will hopefully help if you start to use a project like Yocto and wonder what is going on.

Feel free to put a link to your BitBake project into the comment part below.