# On-Demand Fedora Build Service Documentation

*Release 1*

**Amit Saha**

June 17, 2015

# Contents

**1 Getting Started** 3
- 1.1 Local Mode . . . 3
- 1.2 Distributed Mode . . . 5

**2 Internals** 11
- 2.1 Kickstart files . . . 11
- 2.2 Logging and Build Monitoring . . . 11
- 2.3 In case of Failure . . . 11
- 2.4 Celery integration . . . 12
- 2.5 Zdaemon for Dameonization . . . 12
- 2.6 Deployment . . . 12
- 2.7 Source tour . . . 13
- 2.8 Dependencies . . . 13

**3 API documentation** 15
- 3.1 `bootiso` Module . . . 15
- 3.2 `build_monitor` Module . . . 15
- 3.3 `imagebuilder` Module . . . 15
- 3.4 `notification` Module . . . 16
- 3.5 `repo_create` Module . . . 16
- 3.6 `transfer` Module . . . 16
- 3.7 `util` Module . . . 17
- 3.8 `worker` Module . . . 17

**4 Roadmap** 19

**5 Contact Information** 21

**Python Module Index** 23

**On-Demand Fedora Build service** seeks to build Live and installation images for developers, testers and consumers of Fedora Linux. During the testing of Fedora releases, test images are often useful as smoke tests before full TC/RC composes, as baselines for specific test days or for automated installation testing in AutoQA. The idea is to make an on-demand build service which users/developers can use to make custom Fedora based distributions so that command-line wrangling can be minimized.

CHAPTER 1

# Getting Started

This document aims to get you started with using and deploying the **On-Demand Fedora Build Service**, an effort to make Fedora image building easier. Using this service you can make *boot iso, DVD iso* and *live images* of Fedora Linux.

The service can be used in two modes:

1. Local mode
2. Distributed mode

In *local mode*, all the work is done on the local machine, whereas in *distributed mode*, the build job is distributed across multiple build *nodes* and building images for multiple Fedora releases and architecture is supported.

## 1.1 Local Mode

The *local mode* involves minimum setup requirements and is easy to get started with. If you intend to deploy the service across multiple nodes, the *local mode* is a good way to smoke test the working of the service. Let's get staretd:

1. Clone the source from the git repository using `$ git clone git://github.com/amitsaha/gsoc2012_fbs.git`
2. Install fabric using `$sudo yum install fabric`
3. Navigate to the cloned source repository (`gsoc2012_fbs`). You will see a *fabric fabfile*, `fabfile.py` (basically a symbolic link to `deploy.py` - the real script)

To setup everything you need for running the image building service in local mode, excute `$ fab deploy_local`, which should ask for your `sudo` password as it installs the packages needed for running the image building service, installs the package, `image_builder` and another setup requirement.

Now you are ready to using the image building service. Navigate to the `cli/` directory in the source tree. The file `` `build_cli_basic.py` `` is a command line client to build a Fedora image as per your specifications. Here is a quick overview of how to specify a *boot iso*:

```
[DEFAULT]
type=boot
arch=x86_64
release=17
staging=10.0.0.27
#staging=file:///tmp/staging
email=amitsaha.in@gmail.com

[boot]
product=fedora
```

```
release=17
version=1
updates=0
updates-testing=0
17_url=http://download.fedoraproject.org/releases/17/Everything/x86_64/os
17_mirror=http://mirrors.fedoraproject.org/metalink?repo=fedora-17&arch=x86_64
#17-updates_url=http://download.fedoraproject.org/updates/17/i386
#17-updates_mirror=https://mirrors.fedoraproject.org/metalink?repo=updates-released-f17&arch=i386
proxy=
nvr=package1;packag2;
bid=package3;packag4
outdir=/tmp/lorax_op
workdir=/tmp/lorax_work
```

The `[DEFAULT]` section of the configuration contains the following information:

**`type`** Type of image to be built, has to be one of `boot, dvd, live`

**`arch`** Architecture of the image. Note that, if you are on a `i686` node, you will only be able to build an image of that type. Valid values are `i686, x86_64`

**`release`** The Fedora release of the image to be built. It should be the same as the release of fedora you are running, else unexpected things may happen.

**`staging`** This is where your built images will be stored once completed. There are two options: local file system (specified via ‘`file:///foo/bar`‘) or a FTP server with anonymous access specified via the IP address of the server

**`email`** An email address for image building notification. This isn't active in local mode (although you can activate it using a simple change to the client code, `build_cli_basic.py`‘, which we learn more about a little later).

The `[boot]` section specifies the options which are specific to lorax, Fedora's image creation tool. The `product` option specifies the name of the Boot ISO being created, usually `fedora`. The `release` option is the same as above. The `version` name specifies the version of the ISO being built.

The next few options describe the `repository` and `mirrorlists`. The `updates` and `updates-testing` repository can be enabled/disabled by setting them to `1` or `0`, respectively. The release repository and mirror is specified via the options `{release}_url` and `{release}_mirror` (where `release` is the above specified value). If you have enabled any or both of `updates` or `updates-testing`, you have to specify the repository and mirror list for them as well. Incase you have a `proxy` setup for your repositories, specify it here (**untested**).

Incase you want to include packages which are not yet there in any of the repositories, you may specify them via their `NVR strings` or `Koji build IDs` using the next two options: `nvr` and `bid`. If you don't have any such requirements, just leave them blank. The packages will be pulled in from Koji and included in your image.

Finally, `lorax` requires an output directory where it places the image it builds. You can specify it using `outdir`. The `workdir` option is used to specify a directory which will be used by the image building code to download the extra packages and create a side repository (if any).

Now, you may save the above specification in a file called, `boot_imagebuild.conf`. Next, execute the `build_cli_basic.py` script using `$sudo python build_cli_basic.py /location/for/boot_imagebuild.conf`, which should trigger the build process:

```
Initiating Build Process. See /tmp/imagebuild_13446647457.log for progress
checking for root privileges
checking yum base object
setting up build architecture
setting up build parameters
installing runtime packages
..
```

If you check the specified log file, you should see messages like:

```
2012-08-11 16:07:58,798 - Registered a new Image Build request from amitsaha.in@gmail.com
2012-08-11 16:07:58,798 - Image type:: boot
2012-08-11 16:07:58,800 - Starting the Image Build Process
2012-08-11 16:07:58,800 - Creating side repository
2012-08-11 16:07:58,803 - Downloading packages for Side repository
2012-08-11 16:08:17,350 - Side repository created
2012-08-11 16:08:17,353 - All set. Spawning boot iso creation using lorax.
2012-08-11 16:28:39,101 - Boot ISO built succesfully
2012-08-11 16:28:39,118 - Image building process complete
2012-08-11 16:28:39,118 - Image successfully created. Transferring to staging.
2012-08-11 16:28:39,118 - Initiating local transfer of image(s) to /tmp/staging
2012-08-11 16:28:43,238 - Initiating local transfer of logs to /tmp/staging
2012-08-11 16:28:43,895 - Image(s) and logs available at file:///tmp/staging
```

The `boot_imagebuild.conf` file that was used had the following contents:

```
[DEFAULT]
type=boot
arch=x86_64
release=17
#staging=10.0.0.27
staging=file:///tmp/staging
email=amitsaha.in@gmail.com
[boot]
product=fedora
release=17
version=17
updates=0
updates-testing=0
17_url=http://download.fedoraproject.org/releases/17/Everything/x86_64/os
17_mirror=http://mirrors.fedoraproject.org/metalink?repo=fedora-17&arch=x86_64
#17-updates_url=http://download.fedoraproject.org/updates/17/i386
#17-updates_mirror=https://mirrors.fedoraproject.org/metalink?repo=updates-released-f17&arch=i386
proxy=
nvr=wget-1.13.4-4.fc17
bid=
outdir=/tmp/lorax_op
workdir=/tmp/lorax_work
```

If you compare the logging messages and the image build specification, you will see that I have specified an extra package to be pulled in from Koji using a NVR string and hence the messages regarding side-repository creation.

Similarly, you can also build `DVD` and `Live images` using this command line client and the appropriate specifications. For sample specification for these, see the `conf/` directory in the checked out sources.

It should be kept in mind that **all** the options must be specified in the configuration files as in the samples given. It is therefore recommended that you use one of the sample configuration files to build your own. A simple configuration file generator will soon be made available.

## 1.2 Distributed Mode

The distributed mode enables deployment of the code on multiple nodes and dare we say, in production. The advantages of this mode include support for building images for different Fedora releases and architecture. Also, your local machine will not be hogged with building images. Let's see how we can deploy the build service in this mode.

### 1.2.1 Deployment Configuration

First, the deployment configuration is specified via the file `conf/deploy.conf`:

```
[broker]
i686="amqp://guest@10.0.0.30//"
x86_64="amqp://guest@10.0.0.27//"

[releases]
releases=17,rawhide

[master]
host=gene@localhost
workdir=/tmp/imagebuilder_webapp

[workers-17]
i686=root@10.0.0.30
x86_64=root@10.0.0.27
workdir=/tmp/imagebuilder_worker

[workers-rawhide]
i686=root@10.0.0.37
x86_64=root@10.0.0.43
workdir=/tmp/imagebuilder_worker

[SMTP]
#replace these with appropriate credentials
#single quotes needed
server='smtp.gmail.com'
port='587'
login='bot@gmail.com'
password='f00bar'
```

Let us understand this configuration file's sections and options.

**`broker`** The `url` of the message broker is specified here. This is used by Celery, which is used to distribute the build jobs. As you can see, RabbitMQ is used as a broker in this case. Currently, two brokers are used: one for the `i686` build nodes and another for `x86_64` build nodes. The assumption is that these URLs will point to one of the appropriate worker nodes specified later. For example, the `i686` broker URL should point to a `i686` worker node and similarly for the `x86_64` URL.

**`releases`** Specify the supported releases here.

**`master`** This section specifies the `user@host` string (in the `host` option) of the node which will be the *master* node of the build service. This node will host the Web application that users of this service will use to submit build requests. SSH server should be running on this server to enable deployment and firewall rules appropriately set to allow incoming connections on port 5000. The `workdir` option specifies the directory where the web application will be hosted from.

Next, the build node configurations are specified. Depending on the number of supported releases, there will be one or more sections - one for each supported release, with section names of the form `worker-{release}`, where `release` is one of the values specified in `releases` above.

**`worker-{release}`** This section should specify the `root@host` strings for workers supporting `i686` and `x86_64` image building using the appropriate options. Each worker section should have *both* specified. Multiple `user@host` strings should be seperated via a ; (semicolon). The `workdir` option specifies the directory where the deployment of the build service will take place from.

**`SMTP`** This section specifies the SMTP server configuration which will be used for sending notification emails to the job requester.

### 1.2.2 Deployment using Fabric

Once this configuration file is ready, the fabric script `fabfile.py` will be used to deploy the service. Let us see the tasks that are currently available in the script (assuming you are in the source root):

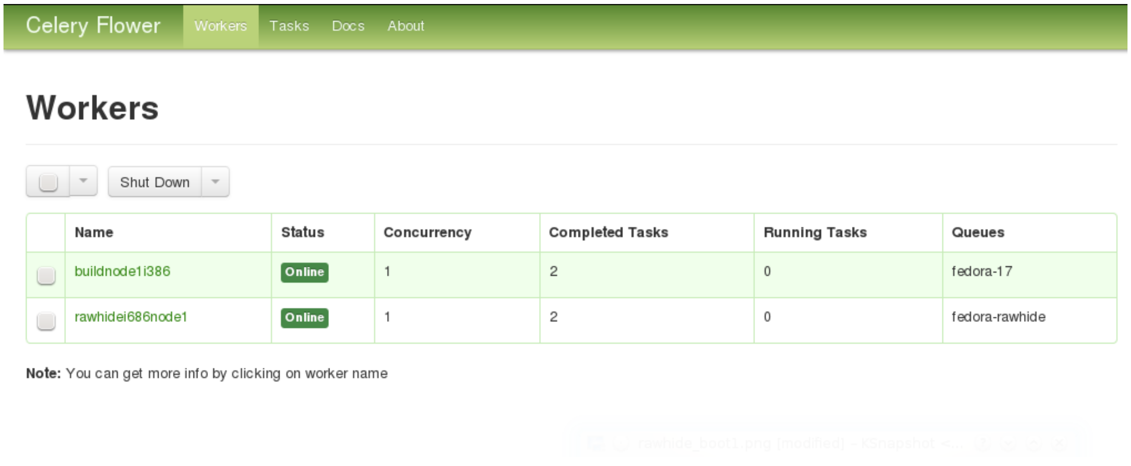
```
$ fab --list

This is a fabfile (http://docs.fabfile.org/en/1.4.2/index.html)
to deploy On-Demand Fedora Build Service.
See doc/ for usage guidelines.

Available commands:

    copy_files_webapp          Copy files to the web application host
    copy_files_workers         Copy the files to the workers
    deploy_local               Deployment in local mode
    deploy_webapp              Deploy the web application (and enable REST API)
    deploy_workers             Deploy the workers
    install_packages_webapp    Install dependencies for the web application
    install_packages_workers   Install dependencies on the workers
    setup_cli                  Deployment for using the command line client in distributed mode
```

### 1.2.3 Deploying the Workers

First, let us deploy the workers. We shall first copy the appropriate files to the worker, install the necessary packages and then finally start the worker processes (root access to all the workers is required). We run the corresponding taks via `fabric`:

```
$ fab copy_files_workers install_packages_workers deploy_workers
[root@10.0.0.37] Executing task 'copy_files_workers'
[root@10.0.0.37] run: rm -rf /tmp/imagebuilder_worker
[root@10.0.0.37] Login password for 'root':
..
..
```

It will take a while before all the taks are completed and provided there are no errors, your workers should now be up and ready to build. Assuming that the above deployment completed without any errors, you can do a simple check to verify whether the workers are up and running using flower (a web-based tool for monitoring celery workers.). In your browser, open the URL `http://<ip>:8008`, where `<ip>` can be any of the worker node IPs. If the IP address you chose is one of the `i686` nodes' IPs, then you should see a webpage showing *all* the `i686` nodes you specified. For example, the following screenshot shows a sample flower interface:



Workers

|  | Name | Status | Concurrency | Completed Tasks | Running Tasks | Queues |
|---|---|---|---|---|---|---|
| □ | buildnode1i386 | Online | 1 | 2 | 0 | fedora-17 |
| □ | rawhidei686node1 | Online | 1 | 2 | 0 | fedora-rawhide |

**Note:** You can get more info by clicking on worker name

If you do not see *all* of the nodes you specified for the specified architecture, something is wrong, and should be investigated.

Now that the workers are deployed, let us now explore the options currently available for submitting an image build task.

### 1.2.4 Command line

A command line client, `build_cli.py` is available in the `cli/` directory of the source tree. Its usage is same as the command line client in local mode, `build_cli_basic.py`. However, before you can start using the command line client you you will need to setup your client computer using the fabric script we used earlier.

From the source tree root, execute `$ fab setup_cli`. You will notice that it installs a few dependencies and also writes a file, `cli/nodes.conf`. This file contains the information regarding the message brokers we discussed earlier. The client will use the message brokers to communicate with the workers. Once this has been completed, you can now submit a new build request:

```
python build_cli.py ../conf/boot_imagebuild.conf
Sending build task to worker
```

If all goes well, you should get an email at the email address you specified in the config file of the form:

```
Your Image Building Request have been submitted. You may monitor
the progress by going  to http://10.0.0.27:5100/log/tmp/imagebuild_134473790944.log.
You will also recieve an email upon completion.
```

As you can see, the build job is being carried out by a worker node with the IP address: `10.0.0.27`. And you can monitor the progress by clicking that link. Once the job has completed, you should get an email saying that your job has been completed and the log messages. If you specified a FTP server to copy the images to, the image should be available there with a timestamp suffixed to the filename. If there was an error in your image build task, the email will tell you the same.

Note:

```
As of now, there seems to be an yet unsquished bug which sometimes
prevents you from seeing the log files. This has something to do
with ``Celery's`` logging and the image builder's logging. In that
case, the best bet is to just wait till you get a job completion
email and check your staging area for the image(s)/logs.
```

Its important that you specify a anonymous writable FTP server as your staging area here, since otherwise you will have to get the image/logs by logging into the worker node.

It is to be noted that you can use this client from any computer which can access the worker nodes.

### 1.2.5 Web and REST API interface

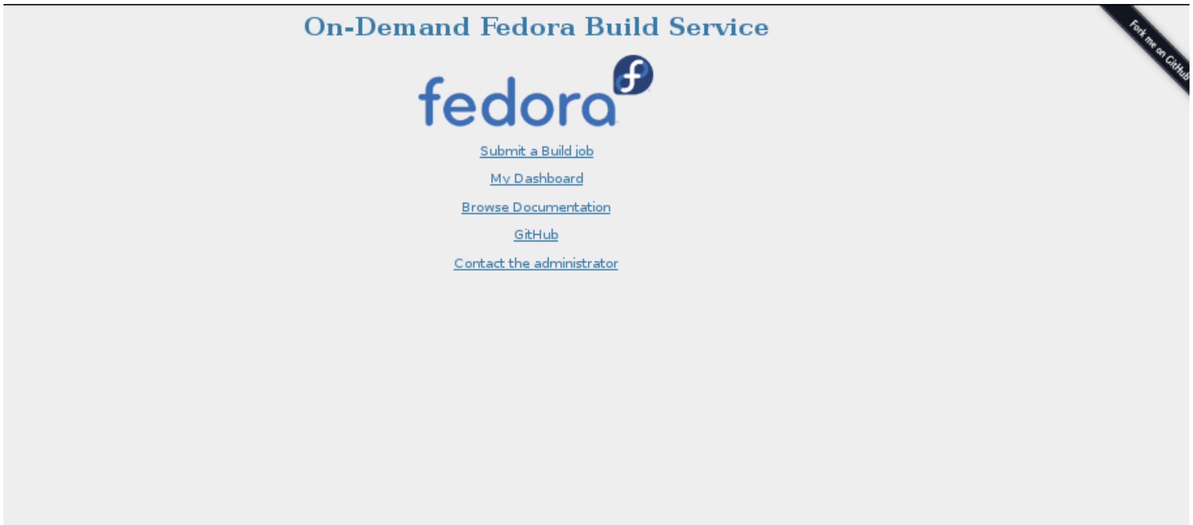
You can also submit your image building jobs via the web application. Let's see how you can set this up. Deploying the web application will involve three steps: copy the files to the web application host computer, install the packages and then finally start the web application. Once again, we use the fabric script to carry these steps:

```
$ fab copy_files_webapp install_packages_webapp deploy_webapp
[gene@localhost] Executing task 'copy_files_webapp'
[gene@localhost] run: sudo rm -rf /tmp/imagebuilder_webapp
[gene@localhost] Login password for 'gene':
[gene@localhost] out: [sudo] password for gene:
```

```
[gene@localhost] run: mkdir -p /tmp/imagebuilder_webapp
[gene@localhost] put: /home/gene/work/gsoc2012_fbs/setup.py -> /tmp/imagebuilder_webapp/setup.py
..
```

Once these steps have been completed successfully without errors, you can now point your browser to `<master>:5000/`, where `<master>>` is the IP address of your web application host as specified in the `deploy.conf` file.

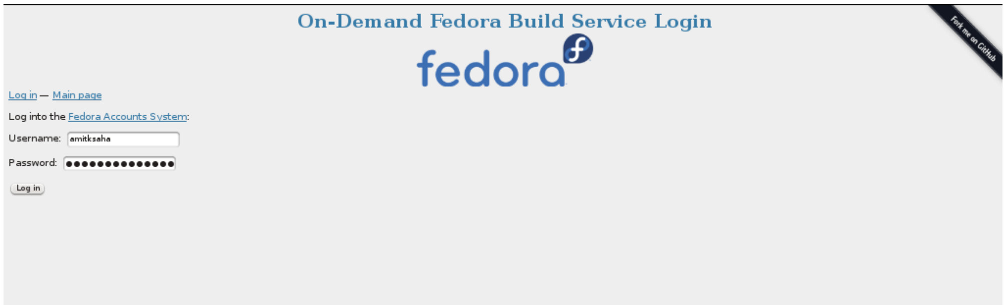
Once you are there, you should see the following interface.



On-Demand Fedora Build Service

The first link allows you to submit a new image build job. Note that it requires you to have a **Fedora Account System** (FAS) login. So, if you don't have one, please create one.

**Login**



On-Demand Fedora Build Service Login

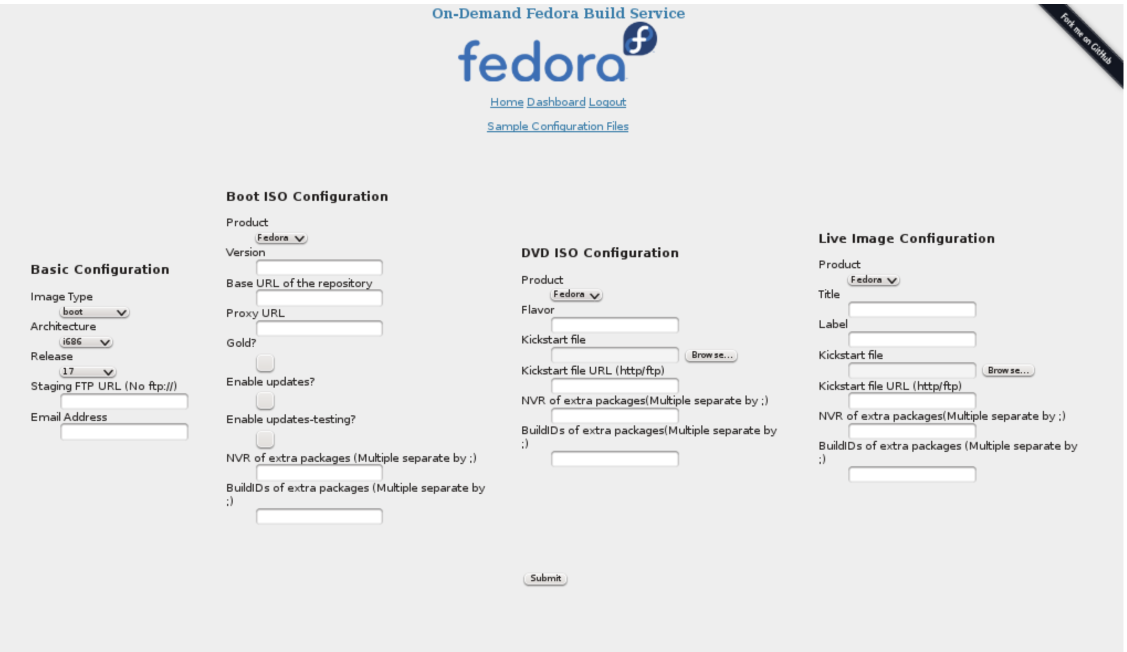
Log in — Main page

Log into the Fedora Accounts System:

Username: amitksaha

Password: ●●●●●●●●●●●●●●●

**Web-based Build Interface**



**Basic Configuration**

Image Type: boot
Architecture: i686
Release: 17
Staging FTP URL (No ftp://)
Email Address

**Boot ISO Configuration**

Product: Fedora
Version
Base URL of the repository
Proxy URL
Gold? □
Enable updates? □
Enable updates-testing? □
NVR of extra packages (Multiple separate by ;)
BuildIDs of extra packages (Multiple separate by ;)

**DVD ISO Configuration**

Product: Fedora
Flavor
Kickstart file
Kickstart file URL (http/ftp)
NVR of extra packages(Multiple separate by ;)
BuildIDs of extra packages(Multiple separate by ;)

**Live Image Configuration**

Product: Fedora
Title
Label
Kickstart file
Kickstart file URL (http/ftp)
NVR of extra packages(Multiple separate by ;)
BuildIDs of extra packages (Multiple separate by ;)

The options on the Web UI are similar to what you specified in the configuration files earlier. You choose the type of image, the architecture, staging, release, etc. Once you hit `Submit`, you should get an email notification similar to the previous section. If you get an email saying *Try again..*, please do so.

The *Dashboard* page which is currently not implemented will have your details of your past and present build jobs submitted.

The web application also exposes a **REST API** endpoint `<master>:5000/rest` which can be then accessed via a REST client to send build requests. An example client is `cli/build_rest.py` which is to be invoked similar to the other command line clients, i.e. `python build_rest.py <config file>`.

**Please note::** The REST API is currently insecure, i.e. there is not integration with FAS. If you consider this a security risk, do not expose this. Simply disable it in the web application, `webapp/app.py`.

Both the web interface and the REST API has the advantage that they allow submitting build requests from any other device having access to the network without any of dependencies required to be installed for the command line client in the previous section.

CHAPTER 2

# Internals

This document will list some of the implementation details and aims to give you a better idea of the project implementation. The details are in no particular order and cannot be considered complete.

## 2.1 Kickstart files

The DVD and Live ISO creation requires the specification of kickstart files. Now, kickstart files are such that one kickstar file may be building upon existing ones. The On-Demand Fedora Build Service recommends you to specify kickstart files which are *ksflattened* via ksflatten. However, there are two possibilities:

1. If you use one of the standard kickstart files, then you don't need to ksflatten them. 2. When you are using the service in local mode, you may specify a kickstart file which includes others, provided they are all in the same directory.

## 2.2 Logging and Build Monitoring

The application uses the standard logging module to log the important events during the image building. A separate log file of the form /tmp/imagebuild_<timestamp>.log is created for each new build request.

In *local mode*, this file can be monitored for the progress and reason out a failure in the process, if any.

In *distributed mode*, this log file is exposed via a *build monitor* whose URL is sent in the email notification on job submission. Basically, this is a Flask web application returning the contents of the log file upon a request.

Here is a note to be kept in mind when the application is deployed in distributed mode:

Due to the integration of the application with *Celery* in the distributed mode, a hackish way has been employed to prevent the application's logging from getting hijacked by *Celery* (which is the default behavior). Now, for some reason yet unknown this doesn't work at random times. In that case, the Build monitor URL sent via the email notification just returns a blank page. Till this is sorted out, it is suggested to ssh to the particular worker node (retrievable from the URL sent) and check the files, /tmp/celery_task.log and /tmp/zdaemon_celeryd.log for progress. If you cannot access the worker node for some reason, just wait for the final notification email and check your staging area for the image/logs.

## 2.3 In case of Failure

The final notification email aims to keep you well informed about the image building result. However, for some reason, you do not get the final email or an email with a blank message (due to the problem cited in the previous section), your best bet is to just contact the person responsible for deployment of the service or if you have access to the worker nodes, simply monitor the two files mentioned in the previous section to look for any possible hints.

## 2.4 Celery integration

In distributed mode, the application uses celery (Celery 3.0) for task delegation to workers using RabbitMQ as the message broker. As of now here is a brief overview of how the task delegation functions right now.

There are different types of supported images that can be built via the application in distributed mode. Different combinations of architecture and release are possible and hence it needs to be ensured that the image building task is picked up by the right worker node. The application maintains two brokers: one for `i686` and another for `x86_64` workers. Thus, the workers capable of building `i686` nodes use the former and `x86_64` capable workers use the latter for listening to incoming requests. Now, each broker additionally has multiple queues - one for each supported Fedora release. That is, the `i686` broker can have a `Fedora 17` queue and a `rawhide` queue, corresponding to the two supported releases. Hence, workers capable of building `i686` images of the `Fedora17` release will be listening to the `Fedora-17` queue on the `i686` broker for incoming jobs and so on. The application takes care of assigning the right queue to a job depending on the incoming build task.

A single celery task is defined in the file `tasks.py`, which takes care of passing the image build job to the workers. A copy of this exist in the `webapp/` as well as `cli/` directory.

## 2.5 Zdaemon for Dameonization

Zdaemon is used to run the various parts of the application as daemon processes: celery, celery flower, build monitor, web application are all run as daemons with the help of `zdaemon`.

## 2.6 Deployment

The deployment of the web application, the worker nodes and pretty much anything to do with the setup of the service is done by a fabric script, `deploy.py`. The script logically defines the complete deployment into a number of *tasks*. It reads the specifications given in the `conf/deploy.conf` configuration file and carries out the specified tasks. See the Getting Started document to see how it is used to deploy the various parts of the application.

The deployment of the *web application* involves two key steps:

- Starting the Web application using the command `python app.py` in the `webapp/` directory. As noted in the previous section, the web application is run as a daemon process using *zdaemon* and hence a configuration file for *zdaemon*, `zdaemon_master.conf` is created by the deployment script and then copied to the host on which the web application is deployed. This configuration file is then used to start `zdaemon`, which in turn starts the web application.
- The second key deployment step that happens is setting up the `celeryconfig.py` module. This module contains the correct configuration of the RabbitMQ message brokers: one for `i686` workers and another for `x86_64` workers. When a new image build request is recieved, the configuration is loaded from this module to delegate the build task to the appropriate broker.

The `deploy_webapp` task carries out the web application deployment task.

The deployment of the workers involve two major steps:

- Starting the RabbitMQ server on each of the two broker hosts. At this stage, one of the workers of each category doubles up as the broker host. The RabbitMQ sever is hence started only on these workers. This is done with the `service rabbitmq-server start` command.

- Starting the celery workers on each of the worker nodes. Depending on the type of image a particular worker is building, it listens to a particular message broker and a queue. Similar to the web application, this deployment is also done by creating a `zdaemon` configuration file and then using `zdaemon` to start the worker processes.

For monitoring purposes, `celery flower` and the image builder's `monitoring` application is also started on each of the workers. The workers are deployed using the `deploy_workers` task.

## 2.7 Source tour

The core of the build service is the package `image_builder`, which lives in the `image_builder/` directory. The `imagebuilder` module is the entry point to the functionalities and invokes the classes and methods defined in the other modules, `worker, notification, transfer` and others.

The Web application lives in the `app/` directory with the `app` module as the entry point.

The command line clients are in the `cli/` directory and the project documentation lives in the `doc/` directory.

The `testing/` directory is supposed to contain `py.test` unit tests for the service, but is currently lacking in them.

The `scripts/` directory contains throwaway scripts not used by any of the other code.

## 2.8 Dependencies

A number of third party software has been used to implement the application. A number of them have been referred to during the course of this document. An exhaustive list can be found from the deployment script, `fabfile.py` in the tasks, `install_packages_webapp` and `install_packages_workers`. Wherever possible, Fedora's package manager is used (i.e. using `yum`) to retrieve the software. However, in some cases, the software is either not packages (for example, Flask-FAS plugin) or the version is outdated compared to the upstream (for e.g. celery 3.0). In such cases, `pip` is used to install them.

CHAPTER 3

# API documentation

The `image_builder` package forms the core functionality of the build service.

## 3.1 `bootiso` Module

**class** `image_builder.bootiso.`**`Bootiso`**(*arch*, *release*, *version*, *repos*, *mirrors*, *proxy*, *outputdir*, *product*)

Create Boot ISO via lorax

**`get_yum_base_object`**(*installroot*, *repositories*, *mirrors*, *proxy*, *tempdir='/tmp'*)

with help from http://git.fedorahosted.org/git/?p=lorax.git;a=blob_plain;f=src/sbin/lorax;hb=HEAD

**`make_iso`**()

Create yum base object and fire the ISO build process

## 3.2 `build_monitor` Module

## 3.3 `imagebuilder` Module

**class** `image_builder.imagebuilder.`**`ImageBuilder`**(*buildconfig*, *kickstart=None*)

**`build`**()

**`checkmonitor`**()

Check if build monitor is running

**`getip`**()

Get the public facing IP address

**`getlogfile`**()

Return the log file

**`initlog`**()

Initiate the logging in local mode

**`notify_email_final`**()

Send a final notification email upon build completion

**notify_email_init**()

Send a notification email upon build initiation with the monitor access URL: <ip>:/log/tmp/imagebuild_xxx.log

## 3.4 notification Module

**class** image_builder.notification.**Notification**

**getconfig**()

**send_email**(*recipient*, *headers*, *message*)

## 3.5 repo_create Module

**class** image_builder.repo_create.**RepoCreate**(*repodir*, *arch*)

Bases: object

Create side repository with additional packages

**download_packages**(*packages*)

Download the packages

**get_build**(*nvr*)

**get_koji_connection**()

Return a Connection to Koji hub

**get_rpm_urls**(*nvr*)

Get the download URLs of the RPMs from NVR string

**get_rpms**(*build_id*)

**make_repo**(*packages*)

Create the side repository after downloading the extra packages

**make_repo_metadata**()

**prep_repo_dir**()

## 3.6 transfer Module

**class** image_builder.transfer.**Transfer**(*buildconfig*, *imgloc*, *logfile*)

Image transfer module

**transfer**()

**transfer_ftp**()

FTP image transfer

**transfer_local**(*staging*)

Local file system copy

## 3.7 `util` Module

Utility functions for the Command line clients

**class** `image_builder.util.`**`Utilities`**

**`get_dict`**(*buildconfig*)

**`get_kickstart`**(*buildconfig*)

## 3.8 `worker` Module

**class** `image_builder.worker.`**`KickstartParser`**

Bases: `object`

**class** `image_builder.worker.`**`Worker`**(*buildconfig*)

Worker class is responsible for the actual creation of the images

**`add_repo`**(*ksfile*, *siderepo*)

Add a repository to an existing KS file

**`build_bootiso`**()

Build boot iso

**`build_dvd`**(*kickstart*)

Builds DVD image

**`build_live`**(*kickstart*)

Live Image

**`gather_repos`**(*release*)

Build repository list using data from the configuration file

**`get_koji_connection`**()

Return a Connection to Koji hub

**`get_nvr`**(*bids*)

get NVR given build ID

**`get_rpms_nvr`**(*nvr*, *bid*)

Take the NVRs and BIDs and return the list of all NVR's

**`prep_siderepo`**(*workdir*, *packages*, *arch*)

prepare a side repository given extra packages

CHAPTER 4

# Roadmap

This project was initiated as part of the **Google Summer of Code‘2012** by *Amit Saha*, mentored by *Tim Flink*.

As of August 13, 2012 - five days away from the project deadline, the service is running on a local infrastructure of multiple workers on different archs and this documentation describes the performance on this infrastructure.

The primary goal of the project was to make it useable by the Fedora QA and that is what we shall be looking at beyond the project deadline. Deployment on Fedora infrastructure is the next major goal.

CHAPTER 5

# Contact Information

Project page: https://github.com/amitsaha/gsoc2012_fbs

Amit Saha: amitksaha@fedoraproject.org

Tim Flink: tflink@redhat.com

# Python Module Index

i

image_builder.bootiso, 15
image_builder.imagebuilder, 15
image_builder.notification, 16
image_builder.repo_create, 16
image_builder.transfer, 16
image_builder.util, 17
image_builder.worker, 17

A

add_repo() (image_builder.worker.Worker method), 17

B

Bootiso (class in image_builder.bootiso), 15
build() (image_builder.imagebuilder.ImageBuilder method), 15
build_bootiso() (image_builder.worker.Worker method), 17
build_dvd() (image_builder.worker.Worker method), 17
build_live() (image_builder.worker.Worker method), 17

C

checkmonitor() (image_builder.imagebuilder.ImageBuilder method), 15

D

download_packages() (image_builder.repo_create.RepoCreate method), 16

G

gather_repos() (image_builder.worker.Worker method), 17
get_build() (image_builder.repo_create.RepoCreate method), 16
get_dict() (image_builder.util.Utilities method), 17
get_kickstart() (image_builder.util.Utilities method), 17
get_koji_connection() (image_builder.repo_create.RepoCreate method), 16
get_koji_connection() (image_builder.worker.Worker method), 17
get_nvr() (image_builder.worker.Worker method), 17
get_rpm_urls() (image_builder.repo_create.RepoCreate method), 16
get_rpms() (image_builder.repo_create.RepoCreate method), 16
get_rpms_nvr() (image_builder.worker.Worker method), 17
get_yum_base_object() (image_builder.bootiso.Bootiso method), 15
getconfig() (image_builder.notification.Notification method), 16
getip() (image_builder.imagebuilder.ImageBuilder method), 15
getlogfile() (image_builder.imagebuilder.ImageBuilder method), 15

I

image_builder.bootiso (module), 15
image_builder.imagebuilder (module), 15
image_builder.notification (module), 16
image_builder.repo_create (module), 16
image_builder.transfer (module), 16
image_builder.util (module), 17
image_builder.worker (module), 17
ImageBuilder (class in image_builder.imagebuilder), 15
initlog() (image_builder.imagebuilder.ImageBuilder method), 15

K

KickstartParser (class in image_builder.worker), 17

M

make_iso() (image_builder.bootiso.Bootiso method), 15
make_repo() (image_builder.repo_create.RepoCreate method), 16
make_repo_metadata() (image_builder.repo_create.RepoCreate method), 16

N

Notification (class in image_builder.notification), 16
notify_email_final() (image_builder.imagebuilder.ImageBuilder method), 15
notify_email_init() (image_builder.imagebuilder.ImageBuilder method), 15

P

prep_repo_dir() (image_builder.repo_create.RepoCreate method), 16
prep_siderepo() (image_builder.worker.Worker method), 17

R

RepoCreate (class in image_builder.repo_create), 16

S

send_email() (image_builder.notification.Notification method), 16

T

Transfer (class in image_builder.transfer), 16
transfer() (image_builder.transfer.Transfer method), 16
transfer_ftp() (image_builder.transfer.Transfer method), 16
transfer_local() (image_builder.transfer.Transfer method), 16

U

Utilities (class in image_builder.util), 17

W

Worker (class in image_builder.worker), 17