ForGe operating and extension manual

Hannu Visti

v1.0

22/9/2013

## Appendix D: Operating instructions

The application is formed of two different components with a different look and feel:

* Database administration interface, administration interface or admin interface is the Django database administration toolkit. It allows manipulation of information stored in the database.
* Application user interface or application is the user interface built for this project. It handles image creation, cheat sheet display and mass upload of files.

### Database administration common parts

The user interface is accessible on Django internal server at

http://localhost:8000/admin/ui

Username and password are the ones set up as administrator passwords when creating the database in Django.

The top part of the page has a ribbon tail and all tables are displayed on the initial landing page. By clicking a table, it displays the table contents. This is customisable; not all database fields are always displayed. To add a new entry, in the upper right corner is a button “add new”. To modify existing, the entry can be selected and changes applied. Entry deletion works by ticking the check boxes before each database item and choosing “delete entries” from the pull down action menu. Entry deletion is a cascading function. It deletes also items that refer to the deleted items. Dates are entered in yyyy-mm-dd format. Time format is hh:mm:ss (24 hour clock). When modifying or creating an entry, if the field header is bolded, it indicates the field must be set and cannot be left empty.

The admin interface allows inserting new items in tables, to which the current editable item refers. For example “secret strategy” refers to case. When opening a secret strategy for editing, the case is shown in a pull down menu. Next to the menu is a green plus symbol. By clicking this symbol the user is taken to the “add new case” form.

### Application user interface common parts

The application user interface main landing page is accessible at

http://localhost:8000/ui/main

The main landing page and all other pages allow navigation by using the links below the application title. Image creation pages have additional case selection in the left sidebar.

### Creating a case

Case is the principal element of a scenario design. It defines the file system type, size and parameters. It defines root directory time and allowed time variance when creating images.

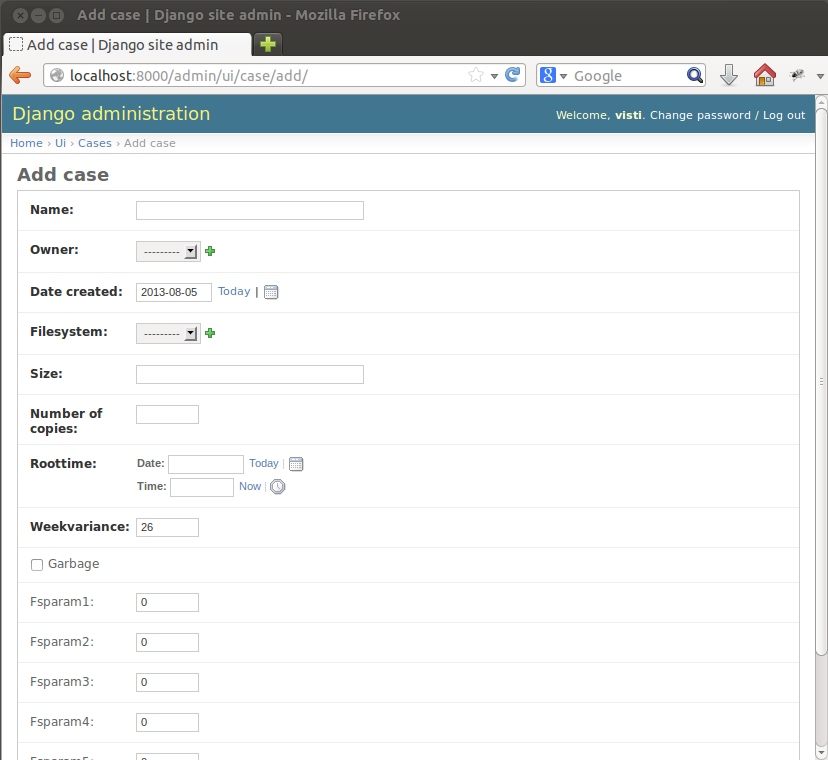


Figure 1 Adding a case

The following fields are available:

* Name: A free form case name. Should use only alphanumeric characters.
* Owner: Must be chosen from pull down menu. Not used in the current version of the application.
* Date created: Default value is today but can be set by user.
* File system: Selected from pull down menu. Only NTFS implemented and available.
* Size: Size of an individual image. Can be an exact number in bytes or can use multiple letters, for example 512K, 3M or 1G.
* Number of copies: The amount of copies requested in the case.
* Root time: Root directory time and date. All MACE timestamps of the root directory itself will be set to this time and date.
* Week variance: The maximum number of weeks the actual time of files and directories on the image can differ from dates and times set as root time, trivial strategy time, secret strategy time or action time. The variance is a signed number; a positive number creates images to the future from set dates and a negative number to the past.
* Garbage: Boolean flag. If set, the image is initially filled with garbage instead of zeroes, before the file system is created.
* Fsparam1: Cluster size in sectors (512 bytes). 8 would indicate cluster size of 4096 bytes.
* Fsparam2-Fsparam5: Not required for NTFS.

### Uploading files

The file upload process is straightforward. The mass upload process happens in the user interface, while a single file can be uploaded in the administration interface. Despite the upload method, the user is advised to verify the automatic categorisation of trivial files. This can be done in the trivial file items part of database administration (Figure 2). Editing file location is not advisable, as it does not move the file to another location and can create a broken reference.

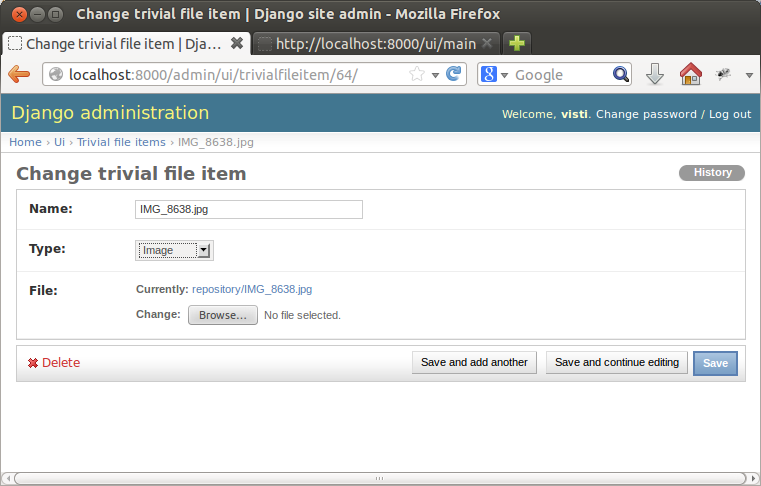


Figure 2 Editing trivial file items

The interesting part is file type, which can be set to any file type from a predefined set. To change the set needs modifying models.py module.

Uploading secret files function in an identical way. Secret file items are not categorised by a type but a group. The group number is an integer identifier. Newly uploaded files get always assigned to group 0. Groups can be modified either individually or directly from the secret file items display. The group eventually links to secret strategies. They point to a group, and at least one file must exist in that group.

### Creating a trivial strategy

The purpose of trivial strategies is to populate the image with files that do not have initially anything secret or conspicuous hidden to them. Some of the trivial items may be used later to hide data to them.

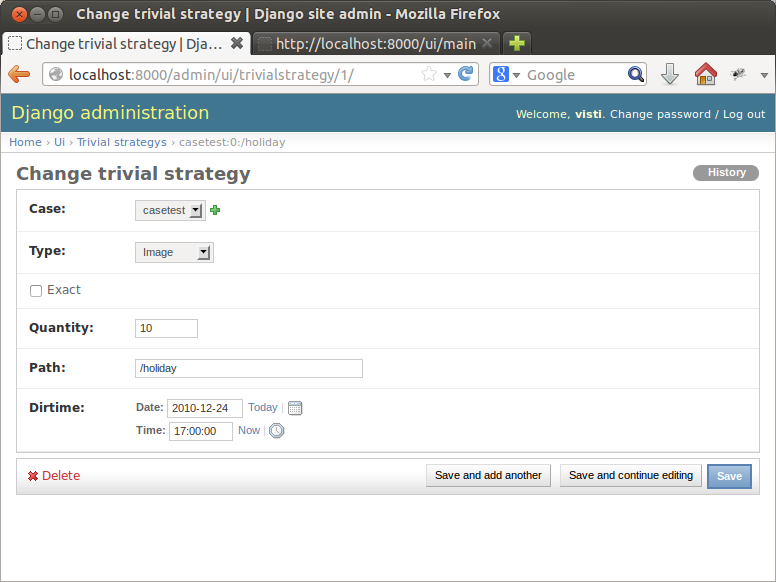


Figure 3 Editing a trivial strategy

The following fields are available (Figure 3):

* Case: The case, to which this strategy applies.
* Type: The file type used in this strategy. Only one can be selected per strategy but several strategies can exist in a single case.
* Exact: A Boolean flag. If the strategy is exact, exactly Quantity number of chosen items appears on the image. If the strategy is not exact, the exact amount is determined randomly for each image between quantity and quantity\*2.
* Quantity: The number of items required. See Exact for explanation.
* Path: The directory, where files are put
* Directory time: Time and date of the directory and files inside. This is modified by case time variance. See section XXX for explanation about trivial strategy timeline processing.

### Creating a secret strategy

Secret strategies direct use of data hiding methods. Figure 4 displays all secret strategies in database sorted by case and Figure 5 illustrates the secret strategy creation/modification field. A secret strategy always hides exactly one file.

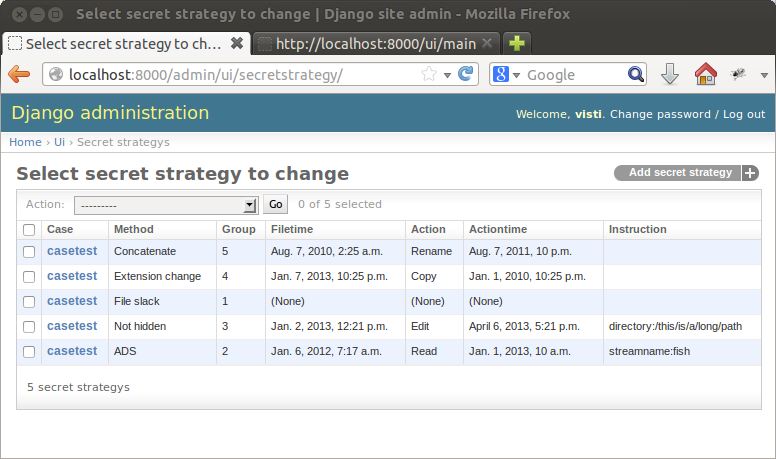


Figure 4 Secret strategies administration

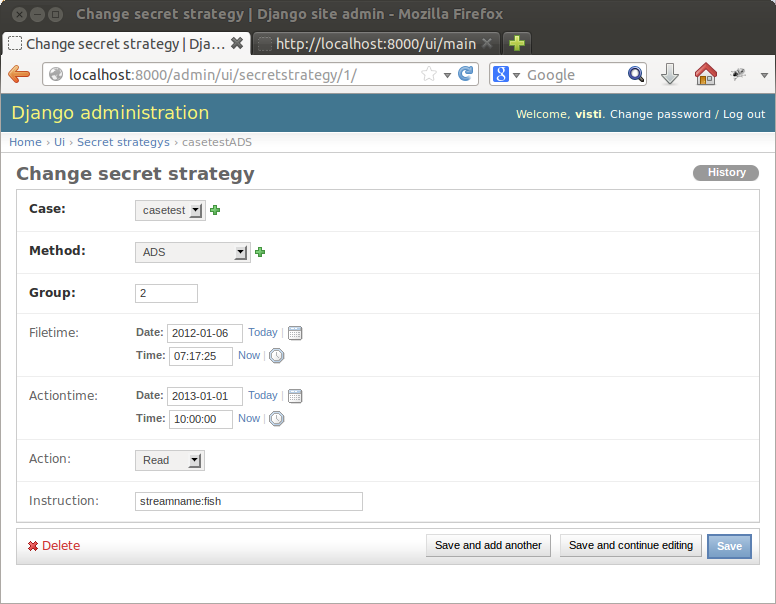


Figure 5 Modifying a secret strategy

When creating a secret strategy, the following fields are available:

* Case: Sets the link to the case, to which this strategy applies. A secret strategy can only apply to a single case.
* Method: The data hiding method the strategy uses. All available methods are displayed in the pull down menu.
* Group: The secret file item group used in this method. If only one file exists in the group, that file is always hidden. If several, exactly one is chosen by random. If no file exist in the group, the image creation fails.
* File time: Optional file timestamp field. If not set, the current system time is applied to the file without time variance modification. If the timestamp is set, this is used as the basis, from which the file time is calculated with the image time variance.
* Action time: If action is set, action time must also be set. This is modified with the time variance.
* Action: Choose from pull down menu. This file operation is simulated by manipulating the timestamps of the file in such a way that it appears this operation has been done to the file at indicated time.
* Instruction: An optional parameter. This depends on data hiding methods. Some of them take optional parameters. Parameters are of form attribute:value attribute2:value2 …

### File systems and hiding methods

These tables need editing only if a new file system or hiding method is added. File system information (Figure 6) is generally a very static table.

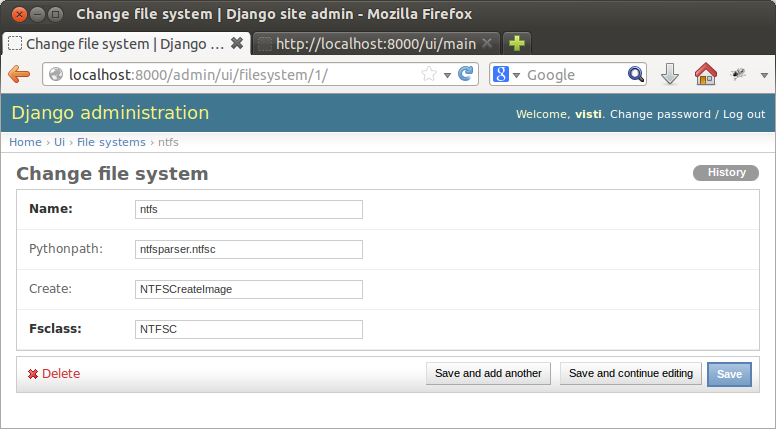


Figure 6 Editing file system information

The following database fields exist:

* Name: Unique name of the file system
* Pythonpath: The path, where the file system class and create function can be found
* Create: A function that creates a file system. This is not part of the class itself, as file system must be created before a class can be instantiated to it.
* File system class: The class that contains all the interface methods.

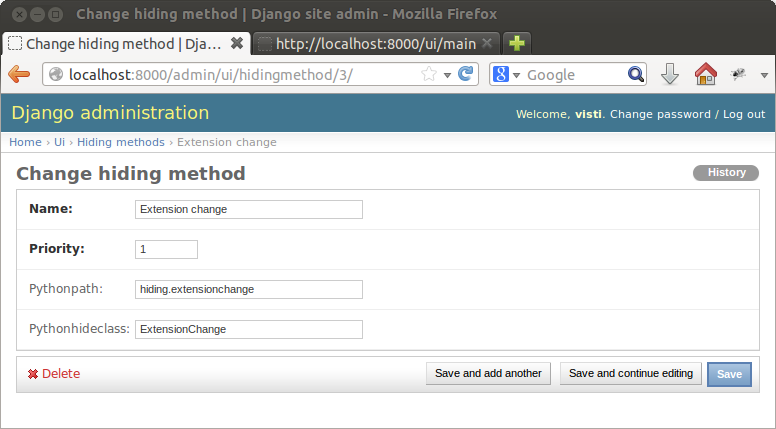


Figure 7 Modifying a hiding method

When creating a new data hiding method, it must be entered to the database. The important parameter is “priority”, which is a user-assigned integer between 1 and 20. When processing secret strategies, they are processed in the order, in which their associated hiding methods are prioritised. Several strategies can have the same priority, in which case their implementation order depends on the order of secret strategy database entries. Hiding strategies assigned to the same priority are not supposed to interfere with each other.

The priority is an interference control method that ensures image integrity. The following priorities are in use in the example application, with explanations:

1. Concatenate, extension change, alternate data streams  
   These are processed first. They depend only on trivial strategy files to function. They can be processed in any order. The result may be that concatenation and alternate data streams target the same file but they do not interfere with each other.
2. Not hidden  
   This is implemented next. The reason to assign it to a lower priority is to ensure the data hiding methods with priority 1 do not choose this file as their target.
3. Deleted file  
   File deletions must occur after all file writes have been completed. This guarantees the MFT record entry of the deleted file remain intact and is not overwritten.
4. File slack  
   Raw disk operations must operate after all writes have been completed. Otherwise data might be overwritten.

The following fields are available:

* Name: A unique name of the data hiding method
* Priority: The priority group. See explanation above.
* Python path: The path that must be included to find the hiding class
* Python hide class: The class that implements the hide\_file() method.

### Creating an image and displaying cheat sheets

Image creation screen requires the user to select a case from the left sidebar and after verifying the details, click create.

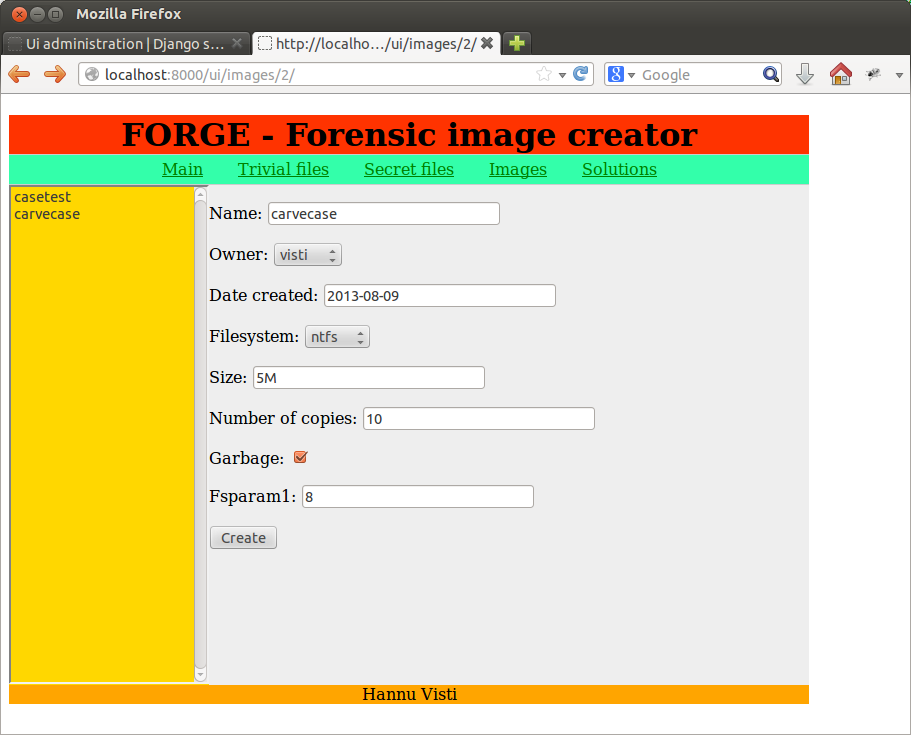


Figure 8 Initiating image creation

The parameters displayed (Figure 8) are for reference only. The fields are editable but they do not affect the image creation. After image creation a result sheet is displayed (Figure 9).

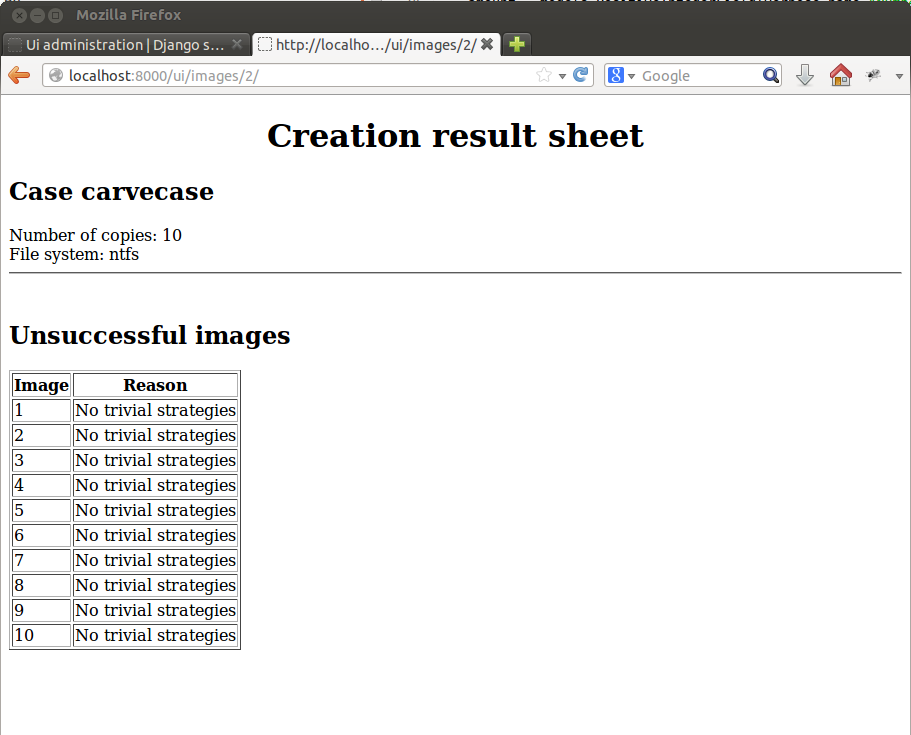


Figure 9 Image creation results

The result sheet is intentionally black and white for easy printing. To navigate away from the page, use the back button of the browser.

Cheat sheet functionality is identical. First, the case is selected, then an image from the pull down menu (Figure 10):

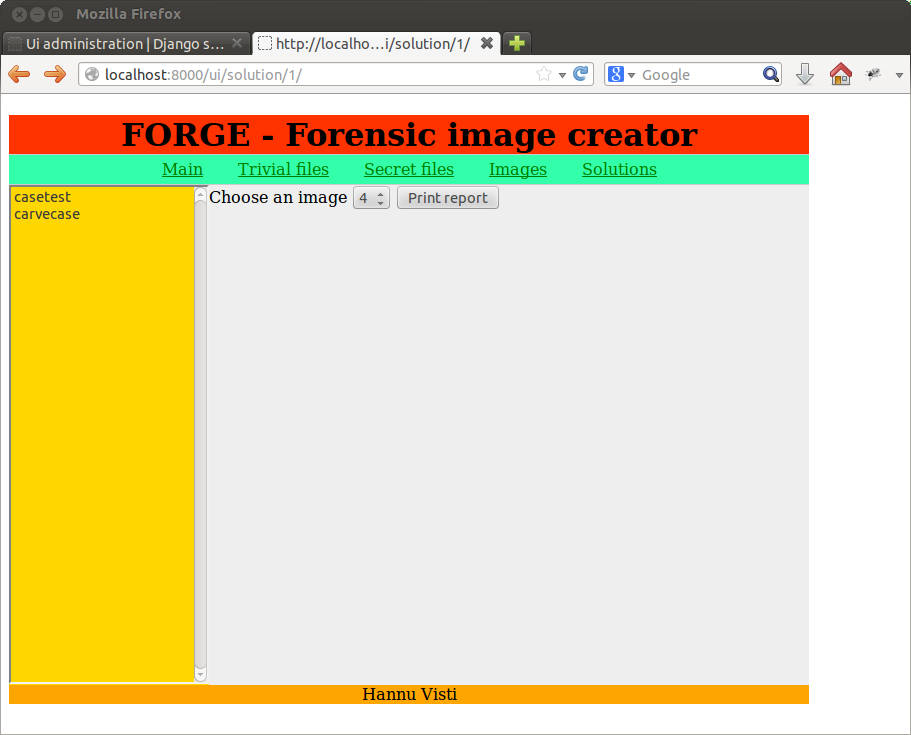


Figure 10 Selecting a cheat sheet to display

The cheat sheet is created based on database entries for that image. If the database has been reset or image data purged, the cheat sheet cannot be created.

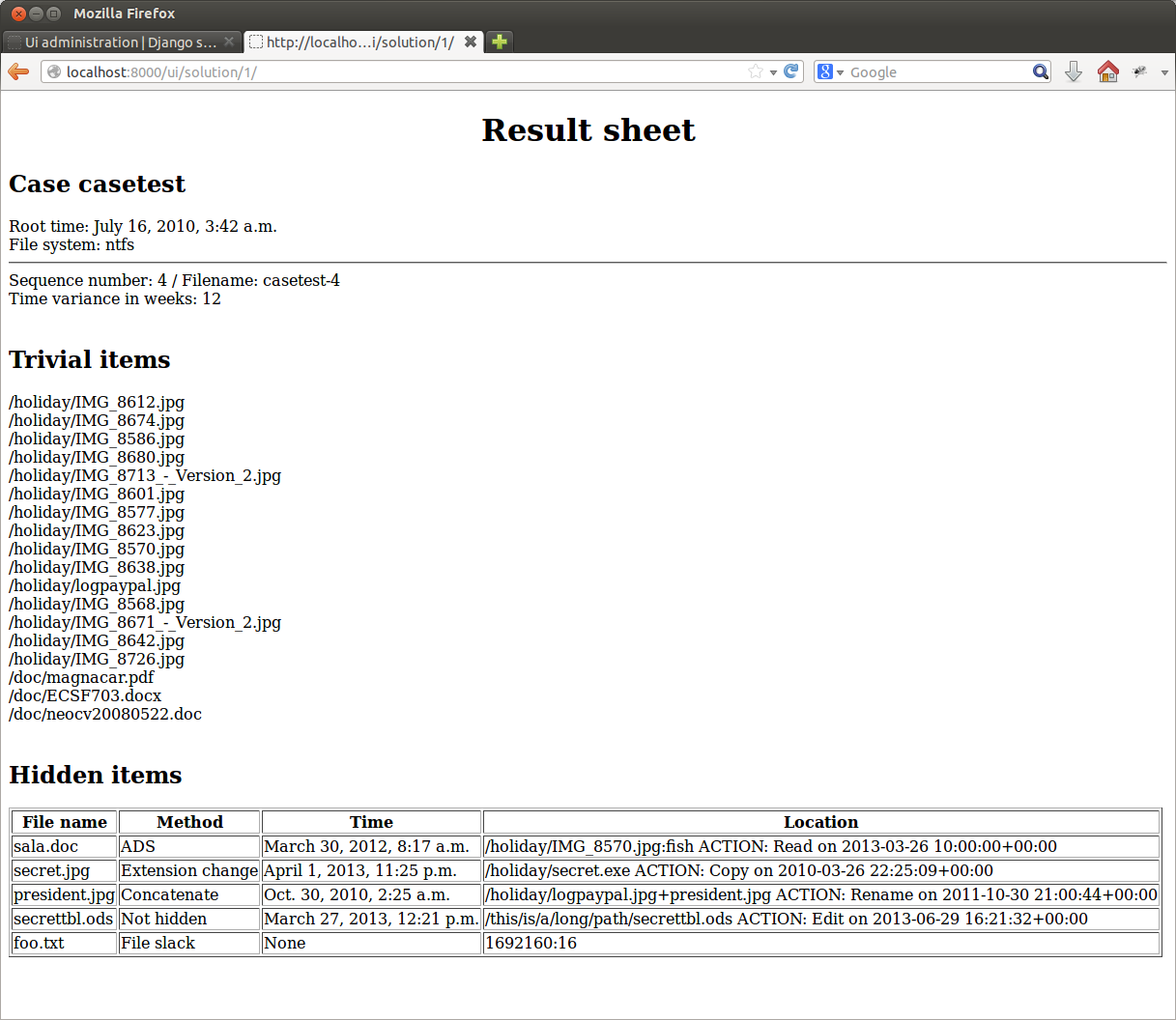


Figure 11 Cheat sheet display

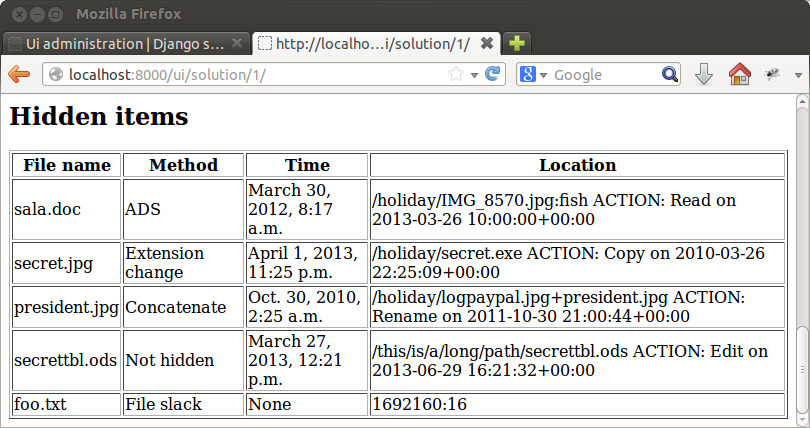


Figure 12 Hidden items highlighted

Figure 11 displays an example of a cheat sheet. Figure 12 is a zoomed-in view to hidden items of the result sheet. The cheat sheet displays case name, image root time and chosen time variance (already implemented in all times displayed on the cheat sheet). Trivial items are listed but their timestamps are omitted as they are supposedly irrelevant to the case.

Hidden items display file name, hiding method, file time and location. If an action was also performed, the location information contains details of the action and action timestamp. Location field format depends on hiding method. The following are in use:

* Alternate data streams: Displays the ADS location, in this example the data is hidden to /holiday/IMG\_8570.jpg:fish.
* Extension change: Original file name is displayed in file name, location displays its new name. In this case the file was named to /holiday/secret.exe
* Concatenate: The files are displayed together. The example result has the file president.jpg appended to /holiday/logopaypal.jpg
* Not hidden: The location of the file is displayed
* Deleted file: The file name with comment DELETED is displayed (not shown here). The same applies to files in concatenate or extension change if a delete request was added to the strategy.
* File slack: start location from image start:number of bytes. If the file was split into several slack spaces, this is a list of location:length location2:length2 location3.length3 …

### Maintenance instructions

The application needs maintenance operations and clean up actions. After image creation, it is user’s responsibility to move the images from the creation directory when they are not needed anymore. It is also the user’s responsibility to clean database of old solution entries when they become obsolete. This is done from the database administration interface under “images”, deleting all images that are not needed anymore. This does not delete image files from the directory but it purges the database of result information. This must be used with care.

Secret and trivial repositories need user maintenance if files are removed. Removing them from the database does not remove the file. This must be done manually. In contrast, removing only the file but leaving the database entry would cause a creation error if the application tried to use the file.

If the database is reset and must be recreated empty, the user can call

http://localhost:8000/ui/init\_db

This sets up basic information to the database. It creates the file system entry, hiding method entries, users, an example case and trivial and secret strategies.

NB: this functionality can be removed by removing the entry from application level URL dispatcher:

urlpatterns = patterns('',

url(r'^$', views.IndexView, name='index'),

url(r'^main$', views.IndexView, name='index'),

url(r'^files$', views.trivial\_file\_view, name='file'),

url(r'^secretfiles$', views.secret\_file\_view, name='file'),

url(r'^init\_db$', views.initDbView, name='file'),

url(r'^posttrivialfile$', views.post\_trivial\_view, name='post'),

url(r'^postsecretfile$', views.post\_secret\_view, name='post'),

url(r'^images/(?P<iid>\d+)/$', views.imageView, name='image'),

url(r'^images$', views.imageView, name='image'),

url(r'^solution$', views.solutionView, name='solution'),

url(r'^solution/(?P<iid>\d+)/$', views.solutionView, name='solution'),

By commenting the bolded line out, the initialisation view becomes inaccessible. The initialisation view uses python paths that work on the development platform. There is no guarantee they work without modification on any other setup but provide a basis for modification.

### Extension programming instructions

#### File system

File system implementation must provide one function and a class. The function creates an image file and the class does everything else.

Creation function

**function(name,size,garbage,clustersize)**name: String/path to the image  
size: String/size of partition. Must accept M/K/G suffix, for example 5M  
garbage: Boolean/True=image initialised full of random characters  
clustersize: Integer/Cluster size

The function name is not predefined by the application but stored in database instead.

The file system class, the name of which is stored in database, must implement the following methods:

Initialisation methods

**\_\_init\_\_(filename, mountpoint)**filename: string/full path name to the image file  
mountpoint: string/full path to the directory where images are mounted  
return value: none

File system class constructor. Attaches an image file to the class and also tells the class the directory where mount and dismount images should operate.

**fs\_init()**  
return value: none

Initialises file system structures. Implementation depends on file system. This function differs from the constructor in the fact that this can be called several times if the application needs to reinitialise file system structures, for example due to changes to metadata or files.

Mounting and dismounting methods

**mount\_image()**  
return value: integer/0=success, other values failure

This method mounts the file system. Implementing this may require setuid programming outside Python.

**dismount\_image()**return value: integer/0=success, other values failure

Dismounts the file system.

File finding methods

**get\_list\_of\_files(flags)**flags: Integer/types of files to be found  
return value: list of FileEntry objects on success, ForensicError raised on failure

Currently implemented flags: 0x1 = system, 0x2 = directory, 0x4 = regular file. A file without flags would be a deleted file or inactive entry. Flags can be ORed together, for example flags 5 would return all regular system files.

**find\_file\_by\_path(path)**path: String/path to file to be located  
return value: a FileEntry object on success, ForensicError raised on failure

Time related methods

**change\_time(path, btime)**path: String/path to file to be located  
btime: Dictionary/time information  
return value: none on success, ForensicError raised on failure

Changes one or several time attributes of a file. Btime is a dictionary that can have the following keys: all, mtime, ctime, atime or etime. The parameter value is a Django DateTime object. Btime can have one or more keys in the dictionary if several time values need to be changed simultaneously. Keyword all is a shortcut to change all time values but can be overridden with a specific parameter. Thus

change\_time(path, dict(all=datetime1, atime=datetime2)

would change all other times to datetime1 except atime, which would be set to datetime2.

**implement\_action(act)**act: Array/act[0] = filename, act[1] = action dictionary  
return value: String indicating action taken. On failure ForensicError raised

This method implements a file action that affects file time attributes depending on file system. Actions include for example read, edit or rename. Action dictionary is of format d[“action”] = datetime, where action is one of the predefined actions (see database table SecretStrtegy)

Raw disk access methods

**get\_file\_slack()**return value: Array/list of slack, None if slack space is not available

The return array on success is formed of elements [start byte, available space, used space] for each slack space chunk. If no slack space is available or the file system does not support slack space hiding, None should be returned. Start byte is absolute from image start, available space is the number of consecutive bytes available in this chunk. Used space registers already used slack space to prevent overwriting.

**register\_used\_file\_slack(location,bytes)**location: Integer/start of the used chunk  
bytes: Integer/Bytes used from this chunk  
return value: none

**write\_location(position, data)**position: Integer/the absolute location where data is written  
data: Array/array of bytes to be written to the location  
return value: None on success, ForensicError raised on failure

#### Data hiding method

Data hiding methods must be classes that implement two methods:

**\_\_init\_\_(filesystem)**filesystem: FileSystemC object/handler to the file system

The constructor takes the underlying file system as a parameter and stores it for data hiding use. This allows access to file system methods to implement data hiding.

**hide\_file(file, param = {})**file: FileField/Django File object pointing to the file to be hidden  
param: Dictionary/data hiding method dependant parameters, defaults to empty dictionary.  
return value: Dictionary/instructions on success, ForensicError raised on failure

The hiding functionality code has access to file system (see constructor) and the file to be hidden. Parameters come in as a dictionary

param[“key”] = value

Parameters are not predefined. Each hiding method can use as many or as few parameters as needed. The author suggests implementing a hiding method in such a way that parameters are not necessarily needed. This can be achieved by providing default values inside the method.