Proposal for GSoC 2021

Scrapy's Feed Enhancements

Personal Details

Name: D R Siddhartha

University: Birla Institute of Technology, Mesra

E-mail: siddharthadr11@gmail.com

Github: github.com/drs-11

Country: India

Abstract

This project aims to add enhancements to Scrapy's Feed Exporter components. These enhancements consists of item filters, feed post-processing and batch delivery triggers. Currently there are no convenient ways to filter items before they can be exported. Item filter feature will provide such ways and interface. A Feed post-processing enhancement will enable plugins such as compression, minifying, beautifier, etc. which can then be added to the Feed exporting workflow. Batch creation was a recently introduced feature but was limited to only item count constraints. Batch Deliver triggers will be able to make the constraints flexible and give more control to user to create batches.

Deliverables

- Item filters which will decide what items must or must not be exported
- Feed post processing feature which will use pluginable components to process scraped data before storing them

- Batch delivery triggers based on certain constraints which can also be user defined

Technical Details

1. **Filter Items**

Solution:

Items can be filtered based on their class type or/and on some certain conditions. For providing maximum flexibility for filtering items, an `ItemChecker` class can be used which can be extended by the user to suit their needs of filtering. This will be a simple implementation which won't break much code(hopefully).

There will be 3 main public methods which will be used in item filtering of which `accepts_item` and `accepts_fields` can be overridden by user. Subdividing `accepts` into 2 methods will help create simple code and will be easier to debug and test.

These filters can be declared in `settings.py`. When the `FeedExporter` is initialised, `load_object` loads the declared filters. These loaded filters can then be used by storage slots. Some common item filtering use cases can be exposed through `settings.py` as well such as accepting certain item classes. Custom item checkers thus can also expose certain feed options in `settings.py`.

Definitions/Examples:

ItemChecker class prototype(This by no means is complete and will probably require

more attrubutes and methods):

```
class ItemChecker:
    accepted_items = []    # list of Items user wants to accept

    def __init__(self, feed_options):
        # 1. populate accepted_items from feed_options if present
        # 2. load items from accepted_items

    def accepts(self, item):
        """
        Main method which will check if item is acceptable or not
        :param item: scraped item which user wants to check if is acceptable
        :type item: scrapy supported items
        :return: `True` if accepted, `False` otherwise
        :rtype: bool
        """
        adapter = ItemAdapter(item)
        return self.accepts_item(item) and
self.accepts_fields(adapter.asdict())

    def accepts_item(self, item):
        """
        Method to check if item as a whole passes filtering criteria. Can be
        overriden by user if they want their own custom filter.
        Default behaviour: if accepted_items is empty then all items will be
        accepted else only items present in accepted_items will be accepted.
        :param item: scraped item
        :type item: scrapy supported items
        :return: `True` if item in accepted_items, `False` otherwise
        :rtype: bool
        """
        if self.accepted_items:
            return isinstance(item, self.accepted_items)

        return True    # all items accepted if none declared in accepted_items

    def accepts_fields(self, fields):
        """
        Method to check if certain fields of the item passes the filtering
        criteria. Users can override this method to add their own custom
        filters.
        Default behaviour: accepts all fields.
        :param fields: all the fields of the scraped item
        :type fields: dict
        :return: `True` if all the fields passes the filtering criteria, else
`False`
        :rtype: bool
        """
        return True    # all fields accepted if user doesn't override and
write their custom filter
```

`settings.py` example:

```
from myproject.filterfile import MyFilter1
from myproject.items import MyItem1, MyItem2

FEEDS = {
    'items1.json': {
        'format': 'json',
        'item_filter': MyFilter1,
    },
    'items2.xml': {
        'format': 'xml',
        'accepted_items': (MyItem1, MyItem2),
    },
}
```

Control Flow:

Loading the declared filter:

```
1. FeedExporter initialises
    i. FeedExporter.filters dict is loaded with (key: uri, value:
filter class) from ITEM_FILTERS
    ii. if no filter declared for uri, load the default item
filter
2. when _start_new_batch is invoked
    i. instance of filter assigned for the slot's uri is created
    ii. slot is initialised with filter as one of the attributes
```

Using the filter:

```
1. item_scraped method is invoked
    i. slots are iterated
    ii. for the given slot, slot.item_checker.accepts(item) is
invoked
        a. if returned True, item is exported
        b. else, item is discarded
```

2. **Feed Post Processing**

Solution:

A post-processing feature can help add more extensions dedicated to before-export-processing to scrapy. To help achieve extensibility, a `PostProcessingManager` can be used which will use "plugin" like scrapy components to process the data before writing it to target files.

The `PostProcessingManager` can act like a wrapper to the slot's storage so whenever a write event takes place, the data is run through the plugins in a pipeline-ish way to be processed and then written to the target file.

A number of plugins can be created but there will be a need to specify the order in which these plugins are used as some won't be able to process the data after it has been processed by another(eg: minifying won't work on a compression processed file). These plugins will be required to have a certain Interface so that the `PostProcessingManager` can use it without breaking down from unidentified components.

Few built-in plugins can be made such as for compressions: gzip, lzma, bz2.

Definitions/Examples:

PostProcessingManager class prototype:

```
class PostProcessingManager:
    """
    This will manage and use declared plugins to process data in a
    pipeline.
    :param plugins: all the declared plugins for the uri
    :type plugins: list
    :param file: target file whose data will be processed before write
    :type file: file like object
    """

    def __init__(self, plugins, file):
        # load the plugins here

    def write(self, data):
        """
        Main method to be used to process a file using all the declared
        plugins.
        :param file: file on which processing needs to be done
        :type file: File-like object
        """

    def close(self):
        """
        Close the target file along with all the plugins.
        """

    def _process_using(self, plugin, data):
        """
        Process the file using the given plugin.
        :param plugin: The plugin that will be used for post-processing on the
file
        :type plugin: PostProcessorPlugin like object
        """

    def _load_plugins(self):
        """
        Loads the plugins from self.plugins using load_object.
        """
```

PostProcessorPlugin class inteface:

```python
class PostProcessorPlugin(Interface):
    """
    Interface for plugins that will be used by PostProcessingManager.
    Mostly will act as an file like object with additional methods for
    processing intermediate data.
    """

    def __init__(self, file):
        """
        Constructor method. Takes in a file like object which will be the
        target file where the processed data will be written to.
        """

    def write(self, data):
        """
        Exposed method which will take data passed, process it and then
        write it to target file.
        :param data: data passed to be written to target file
        :type data: bytes
        :return: returns number of bytes written
        :rtype: int
        """

    def close(self):
        """
        Closes this plugin wrapper.
        """

    @staticmethod
    def process(data):
        """
        This will process the data and return it.
        :param data: input data
        :type data: bytes
        :return: processed data
        :rtype: bytes
        """
```

GzipPlugin example:

```
@implementer(PostProcessorPlugin)
class GzipPlugin:
    COMPRESS_LEVEL = 9

    def __init__(self. file):
        # initialise various parameters for gzipping
        self.file = gzip.GzipFile(fileobj=file, mode=file.mode,
                                  compresslevel=self.COMPRESS_LEVEL)

    def write(self, data):
        return self.file.write(data)

    def close(self):
        self.file.close()

    @staticmethod
    def process(data):
        return gzip.compress(data, compresslevel=self.COMPRESS_LEVEL)
```

settings.py example:

```
FEEDS = {
    'item1.json' : {
        'format': 'json',
        'post-processing': ['gzip'],
    },
    'item2.xml' : {
        'post-processing': ['myplugin','xz'],    # order is important
    },
}


POST_PROC_PLUGINS = {
    'gzip': 'scrapy.utils.postprocessors.GzipPlugin',
    'xz' : 'scrapy.utils.postprocessors.LZMAPlugin',
    'myplugin': 'myproject.pluginfile.MyPlugin',
}
```

Control Flow:

Initialising slot with `PostProcessingManager`:

```
1. FeedExporter is initialised
2. _start_new_batch is invoked
    i. PostProcessingManager instance is created for the given
uri
    ii. the instance is wrapped around the slot's storage
```

3. Batch Delivery Triggers

Solution:

Batch delivery can be simplified and made extensible by creating a class `Batch`. It will contain information about current batch so it can be used to detect when a given contraint has been exceeded and new batch needs to be created.

`BatchPerXItems`, `BatchPerXMins`, and `BatchPerXBytes` can be created as builtins with each inhereting parent class `Batch`. These can be located in `scrapy/utils`.

A parameter value `para_val` is stored in the `Batch` class which will be used to compare against the declared `constraint`. A `para_val` for `BatchPerXItems` can be the number of total items currently in the batch, for `BatchPerXMins` it can be total mins passed since it was created. So some `update` calls to update the `para_val` may not require a `slot_file`.

Desired Batch class can then be activated in `settings.py` with a constraint. To help users create complex triggers they could set the constraint to any builtin type or any arbitrary object to suit their needs. If no constraint is set, it will be pointless to load the specified Batch class. Users can add their own custom Batch class by specifying their class path.

To stop and create a new batch from the Spider itself a signal can be used. This will require the user to create a method `self.trigger_batch(feed_uri)` which will send signal `signal.stop_batch` with the feed's URI as argument which can then be intercepted by `FeedExporter` and appropriately call a method to stop and start a new batch for the specified feed. A new method in `FeedExporter` will be needed to trigger batch delivery as `item_scraped` method is used when an item is scraped.

Definitions/Examples:

Batch class template:

```python
class Batch:
    """
    Batch which will store information for current batches and provides
    suitable methods to check and update batch info.
    :param slot_uri: uri for which batch is being created
    :type slot_uri: string
    :param constraint: a constraint or limit to figure out when a new batch
        must be created.
    :param para_val: a parameter value which will be updated and be compared
        against constraint to control batch creation.
    :param batch_id: id number of the current batch.
    """

    def __init__(self, slot_uri, constraint, para_val=0):
        self.uri = slot_uri
        self.constraint = constraint
        self.para_val = para_val
        self.batch_id = 1

    def update(self, slot_file):
        """
        Updates the parameter value according to stats related to paramter
        value and contraint.
        :param slot_file: slot's file which is a good source of stats
        :type slot_file: File like object
        """

    def should_trigger(self):
        """
        Checks if para_val has crossed the constraint or not.
        :return: `True` if para_val has crossed constraint, else `False`
        :rtype: bool
        """

    def reset_para_val(self):
        """
        Resets parameter value back to its initial value and increments
        self.batch_id. Will be used after closing a batch.
        """
```

settings.py example:

```
from myproject.customclassfile import CustomBatch

{
    'items1.json': {
        'format': 'json',
        'batch_constraint': ('item_count', 10)   # (batch_trigger, constraint)
    },
    'items2.xml': {
        'format': 'xml',
        'batch_constraint': ('byte_size', 100)
    },
    'items3.json': {
        'format': 'json',
        'batch_constraint': (CustomBatch, 10)
    },
}


# builtin triggers
FEED_BATCH_TRIGGER_BASE = {
    'item_count': 'scrapy.utils.feedbatch.BatchByXItems',
    'byte_size': 'scrapy.utils.feedbatch.BatchByXBytes',
    'time_interval': 'scrapy.utils.feedbatch.BatchByXMins',
}
```

FeedExporter modification to receive signal:

```python
class FeedExporter:

    @classmethod
    def from_crawler(cls, crawler):
        # original code
        crawler.signals.connect(exporter.trigger_batch, signals.stop_batch)
        return exporter

    # ...
    # original code
    # ...

    def trigger_batch(self, uri):
        """
        Stops and starts a new batch for the given uri.
        :param uri: uri whose batch will be trigger
        :type uri: string
        """
        # 1. identify the slot fitting uri from self.slots
        # 2. if such slot exists
        #     i. close the slot and remove from self.slots
        #     ii. start a new slot with uri's description and new batch id
        # 3. else, return
```

To trigger batch deliveries from their spiders, users can create their own custom method as follows which passes the uri whose batch will be triggered using the `signal.stop_batch` signal. This can be added as a suggestion in the documentations.

```python
class MySpider(Spider):
    # ...
    # user spider code
    # ...

    def trigger_batch(self, uri):
        self.crawler.send_catch_log(
            signal=signals.stop_batch,
            uri=uri
        )
```

Control Flow:

Batch instance creation:

```
1. FeedExporter is initialised
2. self.batches dict is created
3. Batch instances are loaded from settings
4. Batch instances are saved in self.batches with (key=uri,
value=
   Batch instance) for that uri
```

When using a Batch class as a trigger:

```
1. if an item is scraped
    i. iterate through slots
    ii. if slot accepts the item       // assuming this feature
is implemented after item filter
        a. export the item
        b. call self.batches[slot.uri].update(slot.file) to
update paramter value
        c. call self.batches[slot.uri].should_trigger() to check
if parameter value has exceeded
           constraint
            1. if True
                i. call self.trigger_batch(slot.uri)
                ii. call self.batches[slot.uri].reset_para_val()
            2. else, continue the iteration
    iii. continue slot iteration till complete
```

When using signal as a trigger:

```
1. in spider, logic code calls self.trigger_batch(feed_uri)
2. FeedExporter catches signal.stop_batch and invokes
trigger_batch method
3. trigger_batch method determines which feed's batch to close
using feed_uri argument
4. feed_uri's batch is closed and a new batch is created using
helper functions
```

Timeline

May 17, 2021 - June 7, 2021 (Community bonding period)

- Familiarize myself with Scrapy and Zyte community
- Settle and discuss final design details with mentors
- Discuss implementation plans
- Set up weekly report blog

June 7, 2021 - June 21, 2021: (Week 1-2)

- Start work on Batch Delivery Trigger

- Layout code refactor plan (similar steps will be repeated for other enhancements)
    - Figure out affected code from control flow
    - use proposed designs to plan code refactoring
- Start coding
    - create `Batch` class and associated methods
    - Refactor `FeedExporter` init method to take in Batch classes
    - Modify `FeedExporter` item_scraped method to use Batch classes
    - Add trigger_batch method to `FeedExporter` class
- Write tests

June 21, 2021 - July 5, 2021: (Week 3-4)

- Fix bugs and write documentation (similar steps will be repeated for other enhancements)
    - Analyse side effects due to new code additions
    - Add docstrings to new classes and methods
    - Add entries to official documentation
- Start work on Feed Post Processing
- Layout code refactor plan for Feed Post Processing
- Start coding
    - create PostProcessigManager class and methods
    - create PostProcessorPlugin interface
    - create builtin plugins for compressions
    - Modify FeedExporter and _FeedSlot to use PostProcessingManager

July 5, 2021 - July 19, 2021: (Week 5-6)

- Tests for Feed Post Processing
- Documentation and bug fixes for Feed Post Processing
- Start work on Item Filters
- Plan for code refactor
- **First Evaluations** (July 12-16)

July 19, 2021 - August 2, 2021: (Week 7-8)

- Start coding
    - Create ItemChecker class and methods

    - Modify FeedExporter to use ItemChecker
- Tests for Item Filters
- Documentation and Bug fixes

August 2, 2021 - August 16, 2021: (Week 9-10)

- Polish and optimize new enhancements
- Tie up loose ends
- Improve docs
- Can also be used a buffer period for unforseen circumstances

August 16, 2021 - August 21, 2021:

- Code, project submission
- **Final Evaluations**

Possible Roadblocks

- undesirable side effects of new implementations
- feature additions apart from this project (will require redesigning plans)

Technical Knowledge

- ### Programming Experience:
    - I was introduced to Python in my high school days. Since then I have explored and delved into computer sciences and applications ultimatley and obviously leading me to pursue a degree in Computer Science(currently in pre-final year). Python has been my main language for a long time with me trying my hand on some other languages on the side. I am adept with git and Linux. I have basic understanding of software workflows and API design.

- ### Personal Projects:
    - ngc: a git clone in python with basic functionalities
    - http-server-client: a simple http server and client with multithreading support

- ## Open Source Contributions:
    - openage
    - lxd
    - scrapy
        - https://github.com/scrapy/scrapy/pull/4752
        - https://github.com/scrapy/scrapy/pull/4753
        - https://github.com/scrapy/scrapy/pull/4778

- ## Other Remarks:
    - I have not submitted proposals to any other organisations.

    - My summer vacations will start in May till early July. College will start sometime in mid July. Placement exams may start in mid August. According to the scheduled timeline this shouldn't put me off track from completing the project.

    - I will take responsibilty for the implementations and actively try to fix bugs encountered after the deployments of the code.