Ussd Airflow Documentation

Release 0.0

Francis Mwangi

Oct 03, 2017

Contents

Ussd Airflow is a platform used to create ussd application by defining ussd screens in a yaml file

CHAPTER 1

Getting Started

Getting started with ussd airflow. This assumes you know how ussd works, to learn more how ussd works *How ussd works*

Setup

- Run the following command to install

```
pip install ussd_airflow
```

- Add **ussd_airflow** in Installed application

```
INSTALLED_APPS = [
'ussd.apps.UssdConfig',
]
```

- Change session serializer to pickle serializer

```
SESSION_SERIALIZER = 'django.contrib.sessions.serializers.PickleSerializer
↪'
```

- **Add ussd view to handle ussd request.**
  - To use an existing ussd view that is implemented to handle AfricasTalking ussd gateway

```
from ussd.views import AfricasTalkingUssdGateway

urlpatterns = [
    url(r'^africastalking_gateway',
        AfricasTalkingUssdGateway.as_view(),
        name='africastalking_url')
    ]
```

To use the existing Africastalking ussd gateway and your own ussd screen. Create a yaml file. On the yaml create your ussd screen. Learn more on how to create ussd screen here *Creating ussd screens*. For quick start copy the below yaml

```
initial_screen: enter_name

enter_name:
  type: input_screen
  text: Enter your name
  input_identifier: name
  next_screen: enter_age

enter_age:
  type: input_screen
  text: Enter your age
  input_identifier: age
  next_screen: show_details

show_details:
  type: quit_screen
  text: You have entered name as {{name}} and age as {{age}}
```

Next step add this to your settings. For ussd airflow to know where your ussd screens files are located.

```
DEFAULT_USSD_SCREEN_JOURNEY = "/file/path/of/the/screen"
```

To validate your ussd screen file. Run this command

```
python manage.py validate_ussd_journey /file/path
```

To test the ussd view do this curl request.

```
curl -X POST -H "Content-Type: application/json"
-H "Cache-Control: no-cache"
-H "Postman-Token: 3e3f3fb9-99b9-b47d-a358-618900d486c6"
-d '{"phoneNumber": "400","sessionId": "105","text":"1",
"serviceCode": "312"}'
"http://{your_host}/{you_path}/africastalking_gateway"
```

– **To create your own Ussd View.**

class ussd.core.**UssdView**(***kwargs*)

To create Ussd View requires the following things:

* **Inherit from UssdView (Mandatory)**

```
from ussd.core import UssdView
```

* **Define Http method either get or post (Mandatory)** The http method should return Ussd Request

class UssdRequest(*session_id*, *phone_number*, *ussd_input*, *language*, *default_language=None*, *use_built_in_session_management=False*, *expiry=180*, ***kwargs*)

Parameters

· **session_id** – used to get session or create session if does not exits.

If session is less than 8 we add *s* to make the session equal to 8

· **phone_number** – This the user identifier

· **input** – This ussd input the user has entered.

- **language** – Language to use to display ussd
- **kwargs** – Extra arguments. All the extra arguments will be set to the self attribute

  For instance:

```
from ussd.core import UssdRequest

ussdRequest = UssdRequest(
    '12345678', '702729654', '1', 'en',
    name='mwas'
)

# accessing kwarg argument
ussdRequest.name
```

* **define this varialbe *customer_journey_conf*** This is the path of the file that has ussd screens If you want your file to be dynamic implement the following method **get_customer_journey_conf** it will be called by request object
* **define this variable *customer_journey_namespace*** Ussd_airflow uses this namespace to save the customer journey content in memory. If you want customer_journey_namespace to be dynamic implement this method **get_customer_journey_namespace** it will be called with request object
* **override HttpResponse** In ussd airflow the http method return UssdRequest object not Http response. Then ussd view gets UssdResponse object and convert it to HttpResponse. The default HttpResponse returned is a normal HttpResponse with body being ussd text

  To override HttpResponse returned define this method. **ussd_response_handler** it will be called with **UssdResponse** object.

  class ussd.core.**UssdResponse**(*text*, *status=True*, *session=None*)

  Parameters

  - **text** – This is the ussd text to display to the user
  - **status** – This shows the status of ussd session.

    True -> to continue with the session

    False -> to end the session
  - **session** – This is the session object of the ussd session

Example of Ussd view

```
from ussd.core import UssdView, UssdRequest


class SampleOne(UssdView):

    def get(self, req):
        return UssdRequest(
            phone_number=req.data['phoneNumber'].strip('+'),
            session_id=req.data['sessionId'],
            ussd_input=text,
            service_code=req.data['serviceCode'],
            language=req.data.get('language', 'en')
        )
```

Example of Ussd View that defines its own HttpResponse.

```
from ussd.core import UssdView, UssdRequest


class SampleOne(UssdView):

    def get(self, req):
        return UssdRequest(
            phone_number=req.data['phoneNumber'].strip('+'),
            session_id=req.data['sessionId'],
            ussd_input=text,
            service_code=req.data['serviceCode'],
            language=req.data.get('language', 'en')
        )

    def ussd_response_handler(self, ussd_response):
            if ussd_response.status:
                res = 'CON' + ' ' + str(ussd_response)
                response = HttpResponse(res)
            else:
                res = 'END' + ' ' + str(ussd_response)
                response = HttpResponse(res)
            return response
```

Creating ussd screens

This document is a whirlwind tour of how to create ussd screen.

Strong feature of ussd airflow is to create ussd screen via yaml and not code. This make it easier to give the product owners to design ussd without knowing how to code

In ussd airflow customer journey is created via yaml. Each section in a yaml defines a ussd screen. There different types of ussd and each type has its own rule on how to write ussd application

Common rule in creating any kind of screen **Each screen has field called "type"** apart from initial_screen

The following are types of ussd and the rules to write them.

1. Initial screen (type -> initial_screen)

class ussd.screens.initial_screen.**InitialScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

This screen is mandatory in any customer journey. It is the screen all new ussd session go to.

example of one

```
initial_screen: enter_height

first_screen:
type: quit
text: This is the first screen
```

Its is also used to define variable file if you have one. Example when defining variable file

```
initial_screen:
    screen: screen_one
```

```
    variables:
        file: /path/of/your/variable/file.yml
        namespace: used_to_save_the_variable
```

Sometimes you want to send ussd session to some 3rd party application when the session has been terminated.

We can easily do that at end of session i.e quit screen, But for those scenarios where session is terminated by user or mno we don't know that unless the mno send us a request.

Most mnos don't send notifier 3rd party application about the session being dropped. The work around we use is schedule celery task to run after 15 minutes (by that time we know there is no active session)

Below is an example of how to schedule a ussd report session after 15min

example:

```
initial_screen:
    type: initial_screen
    next_screen: screen_one
    ussd_report_session:
        session_key: reported
        retry_mechanism:
            max_retries: 3
        validate_response:
            - expression: "{{reported.status_code}} == 200"
        request_conf:
            url: localhost:8006/api
            method: post
            data:
                ussd_interaction: "{{ussd_interaction}}"
                session_id: "{{session_id}}"
        async_parameters:
            queue: report_session
            countdown: 900
```

Lets explain the variables in ussd_report_session

- **session_key (Mandatory)** response of ussd report session would be saved under that key in session store
- **request_conf (Mandatory)** Those are the parameters to be used to make request to report ussd session
- **validate_response (Mandatory)** After making ussd report request the framework will evaluate your options and if one of them is valid it would mark session as posted (This is used to avoid double ussd submission)
- **retry_mechanism (Optional)** After validating your response and all of them fail we will go ahead and retry if this field is active.
- **async_parameters (Optional)** This is are the parameters used to make ussd request

2. Input screen (type -> input_screen)

class `ussd.screens.input_screen.`**`InputScreen`**(**args*, ***kwargs*)

This screen prompts the user to enter an input

Fields required:

- text: this the text to display to the user.
- **input_identifier: input amount entered by users will be saved** with this key. To access this in the input anywhere {{ input_identifier }}
- **next_screen: The next screen to go after the user enters** input
- **validators:**

  - text: This is the message to display when the validation fails regex: regex used to validate ussd input. Its mutually exclusive with expression
  - expression: if regex is not enough you can use a jinja expression

    will be called ussd request object text: This the message thats going to be displayed if expression returns False

- **options (This field is optional):** This is a list of options to display to the user each option is a key value pair of option text to display and next_screen to redirect if option is selected. Example of option:

```
options:
 - text: option one
   next_screen: screen_one
 - text: option two
   next_screen: screen_two
```

Example:

```
initial_screen:
  type: initial_screen
  next_screen: enter_height
  default_language: en

enter_height:
  type: input_screen
  text:
    en: |
      Enter your height
    sw: |
      Weka ukubwa lako
  input_identifier: height
  default_next_screen: enter_age
  next_screen:
    - condition: input|int == 60
      next_screen: height_above_60
    - condition: input|int == 30
      next_screen: height_below_30
  validators:
    - regex: ^[0-9]{1,7}$
      text:
        en: |
          Enter number between 1 and 7
        sw: |
          Weka namba kutoka 1 hadi 7

enter_age:
    type: input_screen
    text:
      en: |
        Enter your age
      sw: |
        Weka miaka yako
    input_identifier: age
    next_screen: show_information
    options:
      - text:
          en: back
          sw: rudi
        next_screen: enter_height
```

```
    validators:
        - regex: ^[0-9]{1,7}$
          text:
            en: |
              Only nubers are allowed
            sw: |
              Nambari pekee ndio zimekubalishwa
            default: en
        - expression: ussd_request.input|int < 100
          text:
            en: |
              Number over 100 is not allowed
            sw: |
              Nambari juu ya 100 haikubalishwi

show_information:
    text:
      en: |
        Your age is {{ age }} and your height is {{ height }}.
        Enter anything to go back to the first screen
      sw: |
        Miaka yako in {{ age }} na ukubwa wako in {{ height }}.
        Weka kitu ingine yoyote unende kwenye screen ya kwanza
    type: input_screen
    input_identifier: foo
    next_screen: enter_height

height_above_60:
  type: quit_screen
  text: We are not interested with height above 60

height_below_30:
  type: quit_screen
  text: We are not interested with height below 30
```

3. Menu screen (type -> menu_screen)

class ussd.screens.menu_screen.**MenuScreen**(**args*, ***kwargs*)

This is the screen used to display options to select:

- **text:** This is the text to display to the user.
- **options:** This is a list of options to display to the user each option is a key value pair of option text to display and next_screen to redirect if option is selected. Example of option:

```
options:
 - text: option one
   next_screen: screen_one
 - text: option two
   next_screen: screen_two
```

- **items:** Unlike options where each option has its own screen to redirect in items we have a list of items to display and regardless of the input user will be redirected to one screen.

  Example of items

```
menu_screen_with_item_example:
    type: menu_screen
    text: choose one item
```

```
    items:
        text: "{{key}} for {{value}}"
        value: "{{item}}"
        next_screen: display_option
        session_key: testing
        with_dict:
            a: apple
            b: boy
            c: cat
```

In the above example if will display the following text

```
Choose one item
    1. apple
    2. boy
    3. cat
```

If the user selects "2", that would be translated by the value key, it will result to "b", then "b" will be saved with session_key provided and the user will be directed to the next screen which is display_option.

To reference the selected item, use {{your_session_key}}

•**error_message: (optional)** This is message to display if the user enter the wrong value.

defaults to "Please enter a valid choice."

•option and items are mutual exclusive.

Example:

```
initial_screen:
  type: initial_screen
  next_screen: choose_meal
  pagination_config:
    ussd_text_limit: 90
    more_option:
      en: More
    back_option:
      en: Back


choose_meal:
  type: menu_screen
  text: Choose your favourite meal
  error_message: |
    You have selected invalid option try again
  options:
      - text: food
        next_screen: types_of_food
      - text: fruits
        next_screen: types_of_fruit
      - text: drinks
        next_screen: types_of_drinks
      - text: vegetables
        next_screen: types_of_vegetables
      - text: test pagination
        next_screen: test_text_prompt_pagination


types_of_food:
```

```
  type: menu_screen
  text: Choose your favourite food
  options:
    - text: rice
      next_screen: rice_chosen
    - text: back
      next_screen: choose_meal
    - text: test next screen routing
      next_screen:
        - condition: phone_number == '200'
          next_screen: test_next_screen_routing_one
        - condition: phone_number == '201'
          next_screen: test_next_screen_routing_two

types_of_fruit:
  type: menu_screen
  text: No fruits available choose * to go back
  options:
    - text: back
      next_screen: choose_meal
      input_value: '*'

types_of_drinks:
  type: menu_screen
  text: No drinks available choose 0 to go back
  options:
    - text: back
      next_screen: choose_meal
      input_display: "0 "
      input_value: '0'

rice_chosen:
  type: menu_screen
  text: Your rice will be delivered shortly. Choose 1 to go back
  options:
    - text: back
      next_screen: choose_meal

types_of_vegetables:
  type: menu_screen
  text: Choose one of the following vegetables
  items:
    text: Vege {{ item }}
    value: "{{ item }}"
    with_items: "{{vegetables_list}}"
    session_key: selected_vegetable
    next_screen: choose_quantity

choose_quantity:
  type: menu_screen
  text: Choose vegetable size
  items:
    text: "{{ key }} at Ksh {{ value }}"
    value:  "{{ key }}"
    with_dict: "{{ vegetable_quantity }}"
    session_key: selected_quantity
    next_screen: selected_vegetable
  options:
```

```
    - text: back
      next_screen: choose_meal

selected_vegetable:
  type: menu_screen
  text: >
   You have selected this {{selected_vegetable}}
   and this quantity {{selected_quantity}} at
   {{vegetable_quantity[selected_quantity]}}
  options:
    - text: test_list
      next_screen: test_list_with_native_loop

test_list_with_native_loop:
  type: menu_screen
  text: ""
  items:
    text: "{{item}}"
    value: "{{item}}"
    next_screen: test_explicit_dict_loop
    session_key: alphabet
    with_items:
      - a
      - b
      - c
      - d

test_explicit_dict_loop:
  type: menu_screen
  text: ""
  items:
    text: "{{key}} for {{value}}"
    value: "{{item}}"
    next_screen: test_invalid_jija_variable
    session_key: testing
    with_dict:
      a: apple
      b: boy
      c: cat

# we only support {{ }} jinja variables the otheres will be ingnored
# for now
test_invalid_jija_variable:
  type: menu_screen
  text: Choose one of the following vegetables
  items:
    text: Vege {{ item }}
    value: "{{ item }}"
    with_items: "{%vegetables_list%}"
    session_key: selected_vegetable
    next_screen: choose_quantity


# The screens below are testing pagination
test_text_prompt_pagination:
  type: menu_screen
  text: |
    Ussd airflow should be able to wrap anytext that is larger than the one
```

```
    specified into two screens.
  options:
    - text: next
      next_screen: test_pagination_in_menu_options

test_pagination_in_menu_options:
  type: menu_screen
  text: |
    An example of screen with multiple options that need to be paginated
  options:
    - text: screen_with_both_text_and_menu_options_pagination
      next_screen: test_pagination_in_both_text_and_options
    - text: screen_with_both_text_item_options_pagination
      next_screen: test_pagination_in_both_text_options_items

test_pagination_in_both_text_and_options:
  type: menu_screen
  text: |
    This screen has both large text and options that exceed the limit␣
↪required
    so both the prompt and options will be paginated.
  options:
    - text: go back to the previous screen
      next_screen: test_pagination_in_menu_options
    - text: quit this session
      next_screen: last_screen
    - text: this options will be showed in the next_screen
      next_screen: test_pagination_in_both_text_options_items

test_pagination_in_both_text_options_items:
  type: menu_screen
  text: |
    This screen has both large text, options, items that exceed ussd text␣
↪limit
    part of this text would be displayed in the next screen
  items:
    text: "{{item}}"
    value: "{{item}}"
    next_screen: last_screen
    session_key: testing
    with_items:
      - apple
      - boy
      - cat
      - dog
      - egg
      - frog
      - girl
      - house
      - ice
      - joyce
      - kettle
      - lamp
      - mum
      - nurse
      - ostrich
      - pigeon
      - queen
```

```
      - river
      - sweet
      - tiger
      - umbrella
      - van
      - water
  options:
    - text: quit_session
      next_screen: last_screen

last_screen:
  type: quit_screen
  text: end of session {{testing}}

test_next_screen_routing_one:
  type: quit_screen
  text: screen_one

test_next_screen_routing_two:
  type: quit_screen
  text: screen_two
```

4. Quit screen (type -> quit_screen)

class ussd.screens.quit_screen.**QuitScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

This is the last screen to be shown in a ussd session.

Its the easiest screen to create. It requires only text

Example of quit screen:

```
initial_screen: example_of_quit_screen

example_of_quit_screen:
  type: quit_screen
  text: "Test getting variable from os environmen. {{TEST_VARIABLE}}"
```

5. Http screen (type -> http_screen)

class ussd.screens.http_screen.**HttpScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

This screen is invisible to the user. Its used if you want to make an api call. Its very if you want to make a api call so that you can show the user the results in the next screen.

For instance you can make call for balance check using this screen. And display the balance in the next screen.

Fields used to create this screen:

1. **http_request** This field contains all the fields used to make http request. It contains the following fields:
   (a) **method**
   This is the request method to use. either: get, post, put, delete
   (b) **url** This is the url to be used to make the api call

(c) And all the parameters python request module would accept

you will example below

2. **session_key** In this screen the api call is expected to return json body. The json body is saved in session using this session_key
3. synchronous (optional defaults to true) This defines the nature of the api call. If its asynchronous the request will be made later in celery task.
4. **next_screen** After the api call has been made or been scheduled to celery task ussd request is forwarded to this next_screen

Examples of router screens:

```
initial_screen: http_get_example

http_get_example:
  type: http_screen
  next_screen: http_get_url_query
  session_key: get_response
  http_request:
    method: get
    url: http://localhost:8000/mock/balance
    params:
      phone_number: "{{ phone_number }}"
      session_id: "{{ session_id }}"
    verify: false
    headers:
      content-type: "application/json"
      user-agent: 'my-app/0.0.1'

http_get_url_query:
  type: http_screen
  next_screen: http_post_example
  session_key: get_url_query
  http_request:
    method: get
    url: "http://localhost:8000/mock/balance/{{phone_number}}/"

http_post_example:
  type: http_screen
  next_screen: http_async_example
  session_key: http_post_response
  http_request:
    method: post
    url: http://localhost:8000/mock/balance
    params:
      phone_numbers:
        - 200
        - 201
        - 202
      session_id: "{{ session_id }}"
    verify: true
    timeout: 30
    headers:
      content-type: "application/json"


http_async_example:
  type: http_screen
  synchronous: True
```

```
  next_screen: end_of_http_example
  session_key: http_async_response
  http_request:
    method: get
    url: https://localhost:8000/mock/submission
    params:
      phone_number: "{{ phone_number }}"
      session_id: "{{ session_id }}"

end_of_http_example:
  type: quit_screen
  text: >
    Testing response is being saved in session status code is
    {{http_post_response.status_code}} and balance is
    {{http_post_response.balance}} and full content {{http_post_response.
↪content}}.
```

6. Router screen (type -> router_screen)

class ussd.screens.router_screen.**RouterScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

This screen is invisible to the user. Sometimes you would like to direct user to different screens depending on some status.

For instance you want to show different screen to users who are not registered and a different screen to users who have already registered. This is the screen to create.

Fields used to create this screen:

1. **router_options** This is a list of router option. Each router option has the following fields
   (a) **expression** This is a jinja expression that's is evaluating to boolean It can reference anything in the session and parameters in ussd_request
   (b) **next_screen** This is the screen to direct to if the above expression is true
2. **default_next_screen (optional)** This is the screen to direct to if all expression in router_options failed.
3. **with_items (optional)** Sometimes you want to loop over something until an item passes the expression. In this case use with_items. When using with_items you can use variable item in the expression.

   see in the example below for more explanation

Examples of router screens

```
initial_screen: router_exa_1

router_exa_1:
  type: router_screen
  default_next_screen: default_screen
  router_options:
    - expression: "{{ phone_number == 200|string }}"
      next_screen: 200_phone_number
    - expression: "{{ phone_number == 202| string }}"
      next_screen: 202_phone_number
    - expression: "{{ phone_number in [203|string, 204|string,␣
↪205|string] }}"
      next_screen: sample_router_screen_with_loop
    - expression: "{{ phone_number in [ 206|string, 207|string] }}"
```

```
      next_screen: sample_router_screen_with_dict_loop


200_phone_number:
  type: quit_screen
  text: This number is 200

202_phone_number:
  type: quit_screen
  text: This number is 202

default_screen:
  type: quit_screen
  text: This is the default screen

sample_router_screen_with_loop:
  type: router_screen
  default_next_screen: default_screen
  with_items: "{{ phone_numbers[phone_number]}}"
  router_options:
    - expression: "{{ item == 'registered' }}"
      next_screen: registred_screen
    - expression: "{{ item == 'not_registered'}}"
      next_screen: not_registered

registred_screen:
  type: quit_screen
  text: You are registered user

not_registered:
  type: quit_screen
  text: You are not registered user

sample_router_screen_with_dict_loop:
  type: router_screen
  default_next_screen: default_screen
  with_items:
    phone_number: '207'
  router_options:
    - expression: '{{ key == "phone_number" and value == phone_
↪number}}'
      next_screen: 207_screen

207_screen:
  type: quit_screen
  text: >
    This screen has been routed here because the
    phone number is {{phone_number}}
```

7. Update session screen (type -> update_session_screen)

class ussd.screens.update_session_screen.**UpdateSessionScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

This screen is invisible to the user. Sometimes you may want to save something to the session to use later in other screens.

Fields used to create this screen:

1. **next_screen** The screen to go after the session has been saved
2. **values_to_update** This section defines the session to be saved.

   Inside this section should define the following fields
   (a) **key** the key to be used to save
   (b) **value** the value to store with the key above
   (c) **expression** sometimes you want a condition before you can save data in section

Example:

```
initial_screen: screen_one

screen_one:
  type: update_session_screen
  next_screen: screen_two
  values_to_update:
    - expression: "{{phone_number == 200|string}}"
      key: customer_status
      value: registered
    - expression: "{{phone_number == 404|string}}"
      key: customer_status
      value: not_registered
    - key: aged_24
      value: "{{[]}}"
    - key: height_54
      value: "{{[]}}"

screen_two:
  type: update_session_screen
  next_screen: show_saved_status
  with_items:
    - name: Francis Mwangi
      age: 24
      height: 5.4
    - name: Isaac Karanja
      age: 22
      height: 5.4
    - name: Stephen Gitigi
      age: 20
      height: 5.5
    - name: Wambui
      age: 24
      height: 5.4
  values_to_update:
    - expression: "{{item.age == 24}}"
      key: aged_24
      value: "{{aged_24|append(item)}}"
    - expression: "{{item.height == 5.4}}"
```

```
      key: "height_54"
      value: "{{height_54|append(item)}}"

show_saved_status:
  type: quit_screen
  text: |
    The customer status is {{customer_status}}.
    People aged 24 {{aged_24}}
    People with height 5.4 {{height_54}}
```

8. Custom screen (type -> custom_screen)

class ussd.screens.custom_screen.**CustomScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

If you have a particular user case that's not yet covered by our existing screens, this is the screen to use.

This screen allows us to define our own ussd screen.

To create it you need the following fields.

1. **screen_object** This is the path to be used to import the class
2. **serializer (optional)** This if you want to be validating your screen with specific fields
3. **You can define any field that you feel** your custom screen might need.

EXAMPLE:

examples of custom screen

```
class SampleCustomHandler1(UssdHandlerAbstract):
    abstract = True  # don't register custom classes
    @staticmethod
    def show_ussd_content():  # This method doesn't have to be static
        # Do anything custom here.
        return UssdResponse("This is a custom Handler1")

    def handle_ussd_input(self, ussd_input):
        # Do anything custom here
        print(ussd_input)  # pep 8 for the sake of using it.
        return self.ussd_request.forward('custom_screen_2')


class SampleSerializer(UssdBaseSerializer, NextUssdScreenSerializer):
        input_identifier = serializers.CharField(max_length=100)


class SampleCustomHandlerWithSerializer(UssdHandlerAbstract):
    abstract = True  # don't register custom classes
    serializer = SampleSerializer

    @staticmethod
    def show_ussd_content():  # This method doesn't have to be static
        return "Enter a digit and it will be doubled on your behalf"

    def handle_ussd_input(self, ussd_input):
        self.ussd_request.session[
            self.screen_content['input_identifier']
        ] = int(ussd_input) * 2

        return self.ussd_request.forward(
```

```
            self.screen_content['next_screen']
        )
```

example of defining a yaml

```
initial_screen:
  type: initial_screen
  next_screen: choose_meal
  pagination_config:
    ussd_text_limit: 90
    more_option:
      en: More
    back_option:
      en: Back


choose_meal:
  type: menu_screen
  text: Choose your favourite meal
  error_message: |
    You have selected invalid option try again
  options:
      - text: food
        next_screen: types_of_food
      - text: fruits
        next_screen: types_of_fruit
      - text: drinks
        next_screen: types_of_drinks
      - text: vegetables
        next_screen: types_of_vegetables
      - text: test pagination
        next_screen: test_text_prompt_pagination


types_of_food:
  type: menu_screen
  text: Choose your favourite food
  options:
    - text: rice
      next_screen: rice_chosen
    - text: back
      next_screen: choose_meal
    - text: test next screen routing
      next_screen:
        - condition: phone_number == '200'
          next_screen: test_next_screen_routing_one
        - condition: phone_number == '201'
          next_screen: test_next_screen_routing_two

types_of_fruit:
  type: menu_screen
  text: No fruits available choose * to go back
  options:
    - text: back
      next_screen: choose_meal
      input_value: '*'

types_of_drinks:
```

```
  type: menu_screen
  text: No drinks available choose 0 to go back
  options:
    - text: back
      next_screen: choose_meal
      input_display: "0 "
      input_value: '0'

rice_chosen:
  type: menu_screen
  text: Your rice will be delivered shortly. Choose 1 to go back
  options:
    - text: back
      next_screen: choose_meal

types_of_vegetables:
  type: menu_screen
  text: Choose one of the following vegetables
  items:
    text: Vege {{ item }}
    value: "{{ item }}"
    with_items: "{{vegetables_list}}"
    session_key: selected_vegetable
    next_screen: choose_quantity

choose_quantity:
  type: menu_screen
  text: Choose vegetable size
  items:
    text: "{{ key }} at Ksh {{ value }}"
    value:  "{{ key }}"
    with_dict: "{{ vegetable_quantity }}"
    session_key: selected_quantity
    next_screen: selected_vegetable
  options:
    - text: back
      next_screen: choose_meal

selected_vegetable:
  type: menu_screen
  text: >
   You have selected this {{selected_vegetable}}
   and this quantity {{selected_quantity}} at
   {{vegetable_quantity[selected_quantity]}}
  options:
    - text: test_list
      next_screen: test_list_with_native_loop

test_list_with_native_loop:
  type: menu_screen
  text: ""
  items:
    text: "{{item}}"
    value: "{{item}}"
    next_screen: test_explicit_dict_loop
    session_key: alphabet
    with_items:
      - a
```

```
      - b
      - c
      - d

test_explicit_dict_loop:
  type: menu_screen
  text: ""
  items:
    text: "{{key}} for {{value}}"
    value: "{{item}}"
    next_screen: test_invalid_jija_variable
    session_key: testing
    with_dict:
      a: apple
      b: boy
      c: cat

# we only support {{ }} jinja variables the otheres will be ingnored
# for now
test_invalid_jija_variable:
  type: menu_screen
  text: Choose one of the following vegetables
  items:
    text: Vege {{ item }}
    value: "{{ item }}"
    with_items: "{%vegetables_list%}"
    session_key: selected_vegetable
    next_screen: choose_quantity


# The screens below are testing pagination
test_text_prompt_pagination:
  type: menu_screen
  text: |
    Ussd airflow should be able to wrap anytext that is larger than␣
↪the one
    specified into two screens.
  options:
    - text: next
      next_screen: test_pagination_in_menu_options

test_pagination_in_menu_options:
  type: menu_screen
  text: |
    An example of screen with multiple options that need to be␣
↪paginated
  options:
    - text: screen_with_both_text_and_menu_options_pagination
      next_screen: test_pagination_in_both_text_and_options
    - text: screen_with_both_text_item_options_pagination
      next_screen: test_pagination_in_both_text_options_items

test_pagination_in_both_text_and_options:
  type: menu_screen
  text: |
    This screen has both large text and options that exceed the␣
↪limit required
    so both the prompt and options will be paginated.
```

```
  options:
    - text: go back to the previous screen
      next_screen: test_pagination_in_menu_options
    - text: quit this session
      next_screen: last_screen
    - text: this options will be showed in the next_screen
      next_screen: test_pagination_in_both_text_options_items

test_pagination_in_both_text_options_items:
  type: menu_screen
  text: |
    This screen has both large text, options, items that exceed␣
↪ussd text limit
    part of this text would be displayed in the next screen
  items:
    text: "{{item}}"
    value: "{{item}}"
    next_screen: last_screen
    session_key: testing
    with_items:
      - apple
      - boy
      - cat
      - dog
      - egg
      - frog
      - girl
      - house
      - ice
      - joyce
      - kettle
      - lamp
      - mum
      - nurse
      - ostrich
      - pigeon
      - queen
      - river
      - sweet
      - tiger
      - umbrella
      - van
      - water
  options:
    - text: quit_session
      next_screen: last_screen

last_screen:
  type: quit_screen
  text: end of session {{testing}}

test_next_screen_routing_one:
  type: quit_screen
  text: screen_one

test_next_screen_routing_two:
  type: quit_screen
  text: screen_two
```

9. Function screen (type -> function_screen)

class ussd.screens.function_screen.**FunctionScreen**(*ussd_request: ussd.core.UssdRequest*, *handler: str*, *screen_content: dict*, *initial_screen: dict*, *logger=None*)

This screen is invisible to the user. Its used to if you want to call a function you have implemented.

Its like a complement of http screen. In http screen you make a request to an external service to perform some logic.

This screen on the contrary if the logic that you want to be executed is within your application you use this screen to execute it.

Your function will be called with UssdRequest object. And it should return a dictionary that will be saved in ussd session

Below is the UssdRequest that will be used.

class **UssdRequest**(*session_id*, *phone_number*, *ussd_input*, *language*, *default_language=None*, *use_built_in_session_management=False*, *expiry=180*, ***kwargs*)

Parameters

- **session_id** – used to get session or create session if does not exits.

  If session is less than 8 we add *s* to make the session equal to 8
- **phone_number** – This the user identifier
- **input** – This ussd input the user has entered.
- **language** – Language to use to display ussd
- **kwargs** – Extra arguments. All the extra arguments will be set to the self attribute

  For instance:

```
from ussd.core import UssdRequest

ussdRequest = UssdRequest(
    '12345678', '702729654', '1', 'en',
    name='mwas'
)

# accessing kwarg argument
ussdRequest.name
```

Screen specification

class ussd.screens.function_screen.**FunctionScreenSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

Fields used to create this screen:

1. **function** This is the function that will be called at this screen.
2. **session_key** Once your function has been called the output of your function will be saved in ussd session using session_key
3. **next_screen** Once your function has been called this it goes to the screen specified in next_screen

Examples of function screens:

```
initial_screen: enter_fisrt_number

enter_fisrt_number:
  type: input_screen
```

```
  input_identifier: first_number
  next_screen: enter_second_number
  text: Enter your first number

enter_second_number:
  type: input_screen
  input_identifier: second_number
  next_screen: sum_numbers
  text: Enter your second number

sum_numbers:
  type: function_screen
  default_next_screen: display_odd_number
  function: ussd.tests.utils.sum_numbers
  session_key: sum_results
  next_screen:
    - condition: sum_results|int % 2 == 0
      next_screen: display_even_number

display_odd_number:
  type: quit_screen
  text: "The results was an odd number which is {{sum_results}}"

display_even_number:
  type: quit_screen
  text: "The results was an even number which is {{sum_results}}"
```

Once you have created your ussd screens run the following code to validate them:

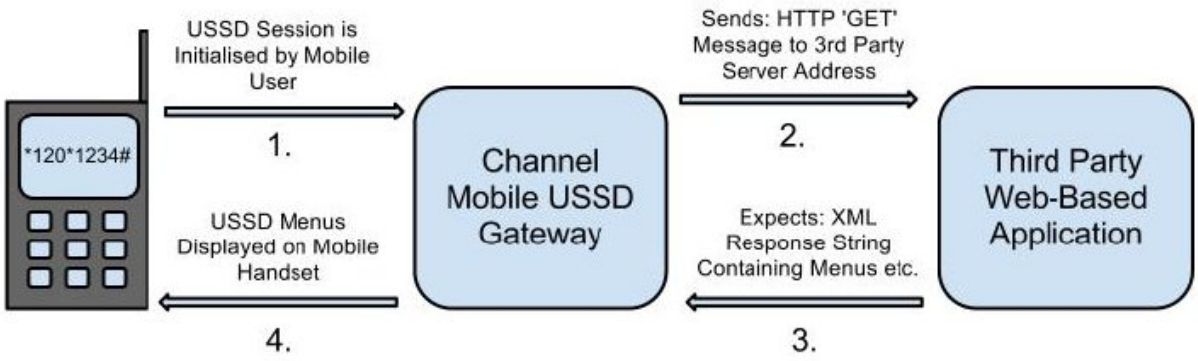
```
python manage.py validate_ussd_journey /path/to/your/ussd/file.yaml
```

CHAPTER 2

How ussd works

How ussd works

Unstructured Supplementary Service Data (USSD) is a protocol used by GSM cellphones to communicate with their service provider's computers. USSD can be used for WAP browsing, prepaid callback service, mobile money services, location-based content services, menu-based information services, or even as part of configuring the phone on the network.



From the diagram above, a request is sent from a mobile phone to a telecom network such Vodafone.

The Ussd Gateway (telecom) then sends the request to your ussd application (i.e where we have the business logic which determines the menu to serve the use on receiving user's request.)

Your ussd application then responds to the request, and Ussd gateway goes ahead and displays your content to the user

Below is a another diagram to help understand the concept

Why Ussd Airflow

Before I explain why we need Ussd Airflow lets first look at one example of ussd user case

Example Menu-Driven USSD Application

One could decide to develop a mobile-initiated "Balance Enquiry and Top Up" application using USSD signaling, enabling a mobile user to interact with an application via the user's handset, in order to view his/her current mobile account balance and top up as needed.

An example of such an application could be as follows:

1. A mobile user initiates the "Balance Enquiry and Top Up" service by dialing the USSD string defined by the service provider; for example, *#123#.
2. TheUSSD application receives the service request from the user and responds by sending the user a menu of options.
3. The user responds by selecting a "current balance" option.
4. The USSD application sends back details of the mobile user's current account balance and also gives the option to top up the balance.
5. The user selects to top up his/her account.
6. The application responds by asking how much credit to add?
7. The mobile user responds with the amount to add.
8. The USSD application responds by sending an updated balance and ends the session.

The figure below shows an example of the MAP/TCAP message sequence required to realize the data transfers between a mobile user's handset and the USSD application to implement the "Balance Enquiry and Top Up" service described above.

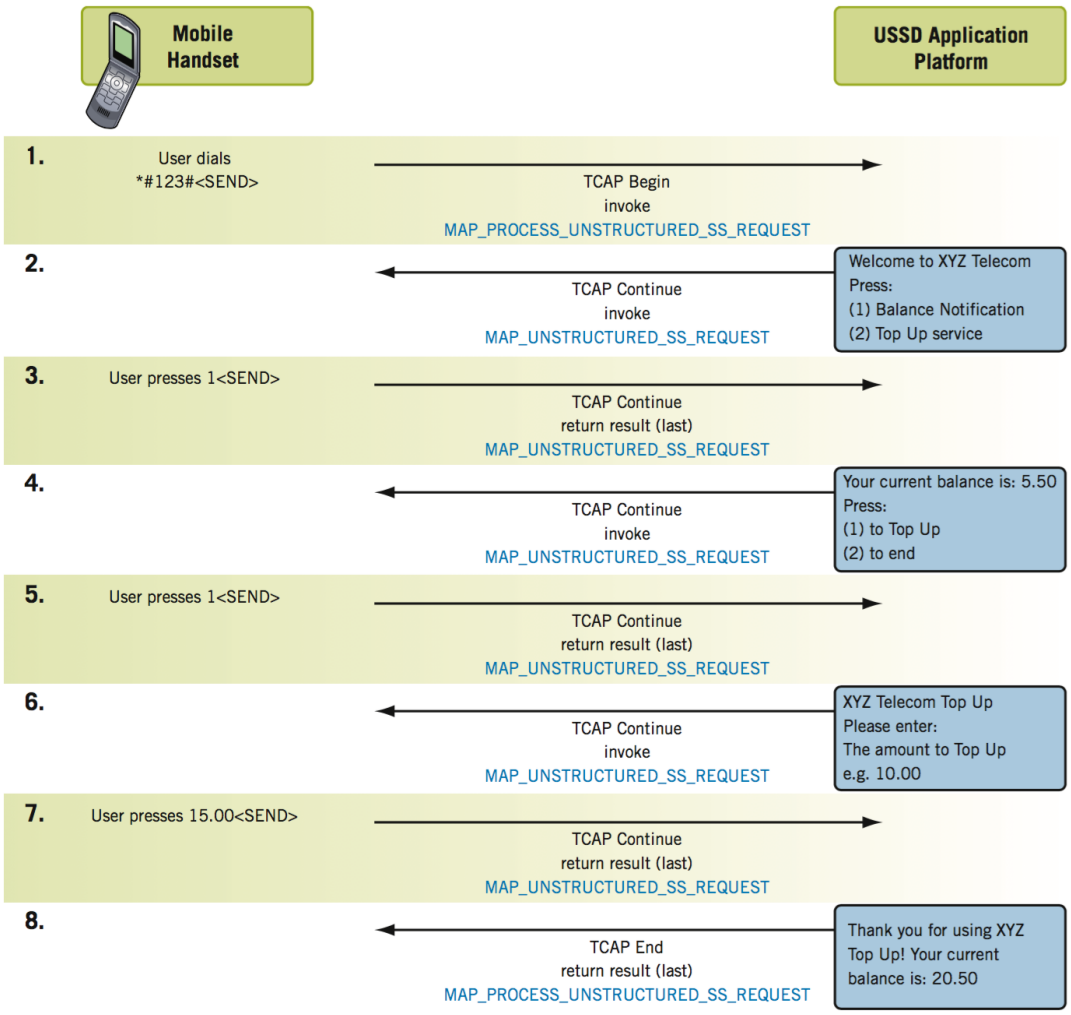


Figure 2. Mobile-Initiated "Balance Enquiry and Top Up" Service (adapted from Figure 6.2, [3GPP TS 24.090])

How ussd airflow comes in

As you have seen in the previous section your ussd application is responsible for the content displayed.

Suppose you want to change the wordings in the ussd screen you are displaying to the user,what is involved in most cases or rather all cases is you make a change in your code and deploy, that's peanuts for most developers

The problem is once you start having many ussd screens and multiple ussd application and many requirements of changing ussd screen, the task that was peanuts becomes overwhelming and would probably start thinking of a way the Product owners would change the ussd content without you being involved and thats where **ussd aiflow** comes in, providing an interface for users to change ussd workflows without code change

CHAPTER 3

Internal Architecture

Comming soon

Python Module Index

u

Index