Down2Earth Telegram bot

Description

The Down2Earth Telegram bot is a proof-of-concept chat bot enabling the delivery of hydrological forecasts to users. The Down2Earth project includes a large-scale hydrological model called CUWALID. The CUWALID model generates output (maps and CSV files) that the chat bot delivers to end users or uses to deliver other forecast formats (respectively). The model forecasts are delivered by this chat bot in a variety of formats including maps, glyphs, text, audio files.

The term "end user" encompasses a wide range of stakeholders and is deliberately not defined here.

This proof-of-concept chat bot is intended to form a demonstrator for the principles of delivering forecast information through interactions with a chat bot and is provided without guarantee or warranty. WhatsApp and Telegram are in use with some of the stakeholder groups but the need for institutional accounts or locally installed WhatsApp servers made the development of a WhatsApp service unfeasible for this research study. Consequently, it was decided to focus on developing a proof-of-concept Telegram bot that could be used to demonstrate the possibilities of delivering forecasts via chat services as a Telegram bot could be created with minimal infrastructure requirements.

This chat bot is provided without guarantee or warranty, see `./LICENSE` .

Please note: this README was originally in Bitbucket and the internal linking format is specific to Bitbucket and may not work in other locations.

Bot operational summary

Telegram users access the chat bot, provide details of their location, choose a forecast, choose a forecast format and this is delivered to them by the chat bot. A flow diagram outlining the operation of the chat bot can be seen in `./chatbot-flow-diagram.png` and viewing the flow diagram while using the bot in Telegram is a good way to understand the operation of the bot.

Language support

The bot can be used in five languages (English, Swahili, Somali, Amharic and Oromo), translations are handled using `gettext` and conventional `.po` and `.mo` files. Further details on how these can be updated to support additional strings or if a translation needs to be changed can be found in `./translation_notes.md` in the same directory as this README.

Technical summary

The chat bot is written in Python and primarily uses the `python-telegram-bot` package (https://pypi.org/project/python-telegram-bot/). The chat bot runs in `long polling` mode (as opposed to using `web`

`hooks` as the latter requires a web server to use) and this is sufficient to demonstrate the functionality of the bot and its efficacy in delivering forecasts.

`python-telegram-bot` effectively uses a state machine to handle the conversation with the user, that is, each point in a conversation is given a state and these states are linked to functions within `./bot/bot.py` and the bot maintains the conversation state the user is in.

The chat bot runs within a `python:3.13-slim` Docker container which should make setup and deployment easier since this would just need a host with Docker running on it, and the data output from the CUWALID model, to build the application. The docker container is set up so the user running the bot is not running in a privileged environment. If running the container in an unprivileged way use the alternate Dockerfile (rename `./docker/Dockerfile.privilege` to `./docker/Dockerfile`).

Please note: the chat bot Docker containers have only been run on Macos and RedHat systems and not on Windows or other Linux distributions.

How to set up the bot

The following are the high level summary tasks required to set up the Wujihacast chat bot. Where appropriate the tasks are linked to the relevant sections in this document for more details.

1. Create Telegram accounts
2. Register/create new bots with @botfather, obtain Bot tokens (see Bot setup)
3. Set up Linux host server with Docker, Docker Compose and a user to run the bot. Current versions in use:-
    - Docker version: 27.4.0
    - Docker Compose bundled version (also tested with 2.31.0 and 2.32.4 standalone)
4. Create directory for code and checkout code on target server
5. Create the relevant `.env` file and populate with Bot token etc (see Environment files)
6. Copy the data files and maps from the CUWALID model to the relevant folders (see Data needed from the CUWALID model)
7. Update the `.env` file to reflect the current year and season
8. Build and run the container (see `./docker/bin/build_test-instance.sh` for test, `./docker/bin/build.sh` for dev and `./docker/bin/build_prod_instance.sh` for production)
9. Run the data checks to confirm that the data and maps files are all present

Data needed from the CUWALID model

CSV data

The chat bot relies on CSV data and maps from the CUWALID model. The CSV model data should be copied into the following location:

```
./bot/data/yyyy/mmm/
```

- `yyyy` - represents the year of the forecast
- `mmm` - represents the period of the forecast (`'mam'`, `'jjas'` or `'ond'`) in lowercase

and the three files should be named as follows:-

- `ET_{MMM}_{yyyy}_county_areas.csv` - Ethiopia data
- `KE_{MMM}_{yyyy}_county_areas.csv` - Kenya data
- `SO_{MMM}_{yyyy}_county_areas.csv` - Somalia data

For example: `ET_OND_2022_county_areas.csv`

The structure of these CSV files is set in the CUWALID model but the first 5 columns are relevant to the chat bot:
`name,place,season,variable,status`

These are:-

1. `name` = A code for the country and the region (for example: `ET_14`)
2. `place` = The name of the region (for example: `Arsi`)
3. `season` = The acronym/code (in upper case) for the season:-
   - `OND` = October-November-December
   - `MAM` = March-April-May
   - `JJAS` = June-July-August-September
4. `variable` = The forecast, possible values are:-
   - `Flood` = Flood Hazard Potential
   - `Pasture` = Pasture/browse Status
   - `Surface` = Surface Water Status
   - `Crop` = Potential crop health
   - `Groundwater` = Groundwater status
5. `status` = The value of the forecast for that region, possible values are:-
   - `0` : Above normal/good
   - `1` : Near normal/OK
   - `2` : Below normal/bad

Flood Hazard Potential status coding

Flood Hazard Potential status is different to the rest of the forecasts as `Above normal` means that there is a higher than normal risk of flooding and this is `bad` (not `good`). This means the data coming from the model via the CSV file uses the opposite coding to the other forecasts. As a result we need to reverse the values before returning the forecast to the user. This is done in the `return_forecast()` function in `bot/bot.py` .

Map images

The CUWALID model will output a map image for each region in the country and for each forecast. The structure of the filename is as follows:-

`[country code]_[region number]_[forecast]_[three month period]_[year]_[language].png`

For example: `ET_15_Pasture_OND_2022_AM.png` is the map for:-

- Ethiopia
- Region number 15 (Bale)
- Pasture/browse Status forecast

- for October-November-December
- of 2022
- Map annotations in Amharic

The map images for the three countries should be copied into (with the same folder structure as the CSV data above):

```
./bot/maps/yyyy/mmm/
```

Data update and checking

When a new forecast is received copy the CSV files into the correct folder and the map files into their folder. Then update the variables in the relevant `.env` file (see Environment files). Once you've done that build a test container and run the following to check that the data is OK and we have all the files expected.

```
cd ./docker
./bin/run_data_check.sh
```

The above script checks:-

1. We have a CSV row for a region in the shapefiles
2. We have a shapefile region for every row in the CSV
3. We have a map file for every region and forecast
4. We have data in the expected format in each row of the CSV files
5. Check we have each of the five forecasts for each region

If the above checks do not pass then the application should NOT be deployed until the data/maps have been fixed!

The data checking script runs checks based on the data provided in the relevant `.env` file. However it's also possible to run the checks by providing command line arguments. These arguments are the `year`, `forecast period` and `instance type`. For example:

```
docker exec –t d2etelegrambot sh –c "cd /srv/projects/d2e/bot/helpers && python check_data.py
2022 ond prod"
```

This would run tests for the year 2022, the forecast period October-November-December and as a prod instance.

Shape files

The bot relies on shapefiles to determine the county in which the user is located. This data is downloaded directly from the ICPAC Geoportal:-

- Kenya - https://geoportal.icpac.net/layers/data0:geonode:ken_adm1 (Admin level 1)
- Ethiopia - https://geoportal.icpac.net/layers/data0:geonode:eth_adm2 (Admin level 2)
- Somalia - https://geoportal.icpac.net/layers/data0:geonode:som_adm1 (Admin level 1)

However, not all of these regions are supported by the CUWALID model and, as of January 2025, the correct region shapefiles have not been supplied so the above files are still in use. However, to address this the bot checks that

forecast data for a specific region is present before trying to deliver the forecast so users will be shown an error saying the region is unsupported in these cases.

Sub-seasonal forecasts

There has been some discussion on providing sub-seasonal forecasts in future. The bot is currently relatively agnostic on the length of the seasons. So, although sub-seasonal forecasts are not currently supported it wouldn't be difficult to add support. The description of the forecast period is relatively loose so you would just need to make sure whatever short string you use for the forecast period in `FORECAST_PERIOD` environment value is matched in the naming of the data folders and relevant entries appear in `forecast_period_lookup`. In addition, you would need to source the relevant translations and update the various `.po` and `.mo` files as outline in `translation_notes.md`. You may also need to make minor updates to the script used to test that the expected data and map files are present.

Bot setup

In order to create a Telegram chat bot you will need to have a Telegram user account (this should be linked to an organisation/business rather than an individual). Once you have an account then you can create a bot using Telegram's `@botfather`. See:-

https://core.telegram.org/bots#how-do-i-create-a-bot

After creating the bot you will receive a token that will be used to register the bot code with Telegram's API and link the bot code to the bot in Telegram.

Bots required

It is recommended that a minimum of three bots are set up in Telegram (for `test`, `dev` * number of developers, `prod`). Ideally the `test` and `dev` bots would be created on Telegram's test instance. However, as of September 2024, authorising test accounts appears to only work reliably if using the iOS client and we do not have access to an iOS device. Although it cannot be tested at this time, if using the test infrastructure the only change required (other than changing the bot username and token values) should be to add `/test` to the end of the bot token (see Environment files), for example:

```
BOT_TOKEN = "bottoken-1234567890abcdefgh/test"
```

As the Telegram's test infrastructure is not supported on Android etc creating test and dev bots on the Telegram production infrastructure seems the best route. Assuming that the `dev` and `test` bots have been created on Telegram's production infrastructure then the bot user names and bot tokens just need to be added to the relevant `.env.[instance-type]` files, see Environment files.

We recommend using separate Telegram accounts for `prod` and `test/dev` to prevent any account issues affecting production. So the arrangement would be:-

```
- Primary Telegram account:
        - Production chat bot

- Secondary Telegram account:
        - Development chat bot
```

```
- Test chat bot
```

Each user account and chat bot will have a separate set of API tokens and the `.env.[instance-type]` should be updated accordingly (see Environment files).

You should **not** use personal accounts for either the *Primary* or *Secondary* Telegram accounts.

Development and test deployment

Deploying the bot in development and testing environments should be done using separate bot instances created within Telegram. That is, a Telegram user creates three bots (for `production`, `development` and `testing`) and these are kept completely separate.

To minimise the risk of bots being run in the wrong context (i.e. a development bot using the production token etc) the build scripts for each instance type use `.env` files that are explicitly named:-

- `.env.dev` - development
- `.env.test` - CI testing
- `.env.prod` - production

and specific build scripts, e.g.:-

```
cd docker
./bin/build_test-instance.sh
```

The other scripts are: `build_dev_instance.sh` and `build_prod_instance.sh`

Environment files

Currently, the bot can be deployed by creating a `.env.[instance-type]` file in the project root and populating with the following:

Filename: `.env.[instance-type]`

```
BOT_TOKEN = "bot-token-from-telegram"
BOT_USER_NAME = "username-created-when-bot-created-at-telegram"
LIMIT_ACCESS = [0|1]
TEST_USER_IDS = "id1,id2"
YEAR = yyyy
FORECAST_PERIOD = 'mmm'
```

Details:-

- `BOT_TOKEN` = Bot token issued from Telegram when bot is created (see Bot setup).
- `BOT_USER_NAME` = Bot user name set when bot is created on Telegram (see Bot setup).
- `LIMIT_ACCESS` = During test or development it may be necessary to limit access to the bot to certain users. Enabling this is done by setting `1`, opening access is set by `0`

- `TEST_USER_IDS` = A comma separated string of Telegram user IDs (not usernames) that are allowed to access the bot *when* access is limited using `LIMIT_ACCESS = 1`. To look up user IDs just add a line to the bot when running in a dev environment to log the `id` parameter on the `user` object (e.g. `message.from_user.id`). If not required set to an empty string.
- `YEAR` = The year of the current forecast (for example: 2024)
- `FORECAST_PERIOD` = The current forecast period (`'mam'`, `'ond'` or `'jjas'`)

Once this is done the Docker container to run the bot can be built and started as using the following commands:

```
cd docker
./bin/build.sh
./bin/start.sh
```

Additional environment variables needed for automated testing

There are several extra fields that are needed in `.env.test` in order to run automated tests:-

```
USER_API_ID = 123456789
USER_API_HASH = 'user-api-hash'
USER_SESSION_STRING = 'user-session-string'
```

`USER_API_ID` and `USER_API_HASH` need to be obtained by logging in to https://my.telegram.org/ and creating a bot user. After that the API ID and API hash will be given on-screen. `USER_SESSION_STRING` needs to be generated and essentially acts as a session authentication token meaning that the tests can interact with the bot without needing to log in each time (which also requires a code from the phone to log in). To generate the `USER_SESSION_STRING` run:-

```
from telethon import TelegramClient
from telethon.sessions import StringSession

USER_API_ID = 1234567890
USER_API_HASH = "YOUR API HASH"

with TelegramClient(StringSession(), USER_API_ID, USER_API_HASH) as client:
    print("Your USER_SESSION_STRING is:", client.session.save())
```

This code is also in a helper function: `./bot/tests/helpers/get_session_string.py`

Once you've generated `USER_SESSION_STRING` add this to `.env.test` and rebuild the container. Treat `USER_API_ID`, `USER_API_HASH` and especially `USER_SESSION_STRING` as **secret values** as anyone obtaining them (particularly `USER_SESSION_STRING`) can access your user account without authenticating. This is another reason why you should not use a genuine user account and create a separate one for these purposes.

Once this is done the Docker container to run the bot can be built and started as using the following commands:

```
cd docker
./bin/build_test-instance.sh
```

Testing

The tests use `pytest` (https://pypi.org/project/pytest/) and include traditional unit tests and End-to-End (E2E) tests. In the case of the E2E tests these use `telethon` to interact with a running instance of the bot.

The tests are marked with three pytest `marks` (see `./bot/tests/pytest.ini`) that allow us to run specific subsets of tests if required:-

- `asyncio` : Test is asynchronous
- `e2e` : E2E tests
- `unit` : Unit tests

Unit tests

To run the tests use:-

```
cd docker
./bin/run_unit_tests.sh
```

E2E tests

The setup for E2E tests is more complicated. E2E tests rely on having access to the Telegram API or mocking all the functions of the Telegram API. The latter is impractical so one solution is to set up a specific/separate instance of the bot and to use the `telethon` (https://pypi.org/project/Telethon/) to programatically interact with the running bot instance.

It is important to separate the bot instance used for production from that used for testing. See Bots required

```
cd docker
./bin/run_e2e_tests.sh
```

The E2E tests have a customisable wait time between each sending of a message. If the time is too short then Telegram will rate limit the user and not accept any more messages from that user for a period of time. This restriction can be around 30 mins but can increase to much longer if the user continues to exceed limits. Currently the wait time is set at 10 seconds but the defaults can be changed in `./tests/helpers/custom_wait.py` and overridden within individual tests using `bot_wait(15)` where the value is the number of seconds to wait.

Despite the custom wait time I was still encountering rate limiting fairly regularly when running tests. To avoid this I ran the test files individually rather than letting them all run sequentially.

Run all tests

```
cd docker
./bin/run_all_tests.sh
```

CI/CD

CI/CD deployment pipelines can be supported by building an instance of the bot in a test docker container (make sure you use a test Telegram account), run unit and E2E tests on the container and then deploy updated containers into production.

Python dependencies

During the build process the required third-party libraries are installed from commands in the `Dockerfile` using `pip`. Different instance types (`prod`, `dev` and `test`) use different sets of requirements as both `dev` and `test` benefit from access to modules that are not needed when running in production (such as `pytest`). The different sets of requirements are set out in separate `requirements.txt` files:-

- `./bot/requirements/prod_requirements.txt` - Python dependencies used in production
- `./bot/requirements/requirements.txt` - Python dependencies used in `dev` and `test` environments

These `requirements.txt` files are compiled from `*.in` files within the same directory so that `prod_requirements.in` specifies everything needed for production but `dev_requirements.in` only lists the additional packages needed for `dev` and `test`.

The two `requirements.txt` files are compiled using the `pip-compile` command and this can be run from:-

```
cd ./docker
./bin/compile_requirements.sh
```

Please note that `./bot/requirements/prod_requirements.txt` and `./bot/requirements/requirements.txt` MUST NOT be edited by hand. Edit the `*.in` files and recompile. The compilation process makes sure all packages are explicitly listed with version numbers and hashes. Hashes are then confirmed during the `pip install` process to protect against supply chain attacks.

Checking for vulnerabilities in dependencies

To check the Python packages for vulnerabilities we have `pip-audit` installed in `dev` and `test` instances. There is a helper script to run checks:-

```
cd ./docker
./bin/check_for_vulns.sh
```

If vulnerabilities are found then update the packages in the relevant `*.in` file, recompile the requirements files, re-run the vulnerability check to confirm the fix and, if all tests pass using the new packages, redeploy the containers.

Production deployment and operation

This chat bot is provided without guarantee or warranty (see `./LICENSE`) as it has not been run in production and has only been used for limited user testing to date. This means that some of the tasks that we would undertake at the University of Bristol prior to production deployment have not been done (for example, performance, load or security testing).

As the bot has not been run for any significant period of time (each test session was only a few hours) it is not known what problems might be encountered with the long-running operation of the bot. However, given the transactional nature of chat bot usage we would expect that periodic restarts of the container would improve operation and have minimal impact on users (but please note issues related to chat persistence when restarting containers in the section on Chat persistence).

Running the bot in production is identical to running the bot in test but (other than configuring for the correct forecast season) with relevant configuration settings:-

```
TEST_USER_IDS = ""
LIMIT_ACCESS = 0
INSTANCE = 'prod'
```

It is assumed that the chat bot containers will be deployed and built using an automated CI/CD pipeline (orchestrated by something like Jenkins). The `.env` file will need to be in place on the target server, code checked out and then the build script (`./docker/bin/build.sh`) run before the container started (`./docker/bin/start.sh`).

Operational tasks will include monitoring the bot's dependencies for vulnerabilities (see Checking for vulnerabilities in dependencies) and updating the third-party packages as necessary, running tests and redeploying.

Chat persistence

Currently there is no chat persistence, this means that if the bot is restarted then the *current state of all chats will be lost*. If a user is part-way through a chat at this point then, whatever their next action, they will return to the start of the chat process (i.e. the option to select their language) and need to start again.

Chats are not currently persisted because, to do so would require (amongst other things) storage of the chat ID for each chat and this is likely to be considered personal data since it is directly tied to an interaction with an individual's Telegram account. This means permission will probably be required from the user in order to store this information according to Data Protection legislation and there are other obligations on the organisation storing the data (for example, right to deletion etc).

The Python Telegram Bot package that we use does have support for chat persistence. The following page should act as a good starting point to add chat persistence at a later date once the issues around getting and tracking permission have been resolved:-

https://github.com/python-telegram-bot/python-telegram-bot/wiki/Making-your-bot-persistent

Ending a chat conversation

Without chat persistence it's impossible to end a chat conversation unless it is ended immediately after the forecast has been delivered, which is undesirable. For example, it might be desirable to end the conversation if there are no further messages from a user for a certain period of time. For this to happen the chat ID has to be stored and this results in the same issues as above.

However, the Python Telegram Bot package has support to end the chat (or in fact perform any form of action) after a period of time. To set a timeout on a conversation then you can use the `conversation_timeout` parameter:

https://docs.python-telegram-bot.org/en/stable/telegram.ext.conversationhandler.html#telegram.ext.ConversationHandler.params.conversation_timeout

For the `conversation_timeout` parameter to be used you also need to install the Python Telegram Bot package with `job-queue` being set and the dependencies installed. This means that the `prod_requirements.in` file will need to be updated to replace `python-telegram-bot` with `python-telegram-bot[job-queue]`, recompiled and the containers rebuilt.

Logging

The bot will log usage to a `bot-log` volume but currently this is limited to very basic usage information. Since the bot interacts with Telegram APIs and not the users directly we don't have the standard logging information you'd expect from, say, a web server. This means that anything we want to log has to be logged explicitly. At this time no data privacy information is provided to users in the test bot so no identifiable information is logged and this means the logs are not useful to trace a specific user's interaction. Once the data privacy issues are known and users can be informed the logging can be updated to include an identifier such as chat ID.