EvalAI Documentation

Release 1.1

CloudCV Team

Jun 09, 2020

Contents

Contents:

How to setup

EvalAI can run on Linux, Cloud, Windows, and macOS platforms. Use the following list to choose the best installation path for you. The links under *Platform* take you straight to the installation instructions for that platform.

1.1 Installation using Docker

We recommend setting up EvalAI using Docker since there are only two steps involved. If you are not comfortable with docker, feel free to skip this section and move to the manual installation sections given below for different operating systems. Please follow the below steps for setting up using docker:

1. Get the source code on to your machine via git

```
git clone https://github.com/Cloud-CV/EvalAI.git evalai && cd evalai
```

2. Build and run the Docker containers. This might take a while. You should be able to access EvalAI at `localhost:8888`.

```
docker-compose up --build
```

1.2 Ubuntu Installation Instructions

1.2.1 Step 1: Install prerequisites

- Install git

```
sudo apt-get install git
```

- Install postgres

```
sudo apt-get install postgresql libpq-dev
```

• Install rabbitmq

```
echo 'deb http://www.rabbitmq.com/debian/ stable main' | sudo tee /etc/apt/sources.
→list.d/rabbitmq.list
sudo apt-get update
sudo apt-get install rabbitmq-server
```

• Install virtualenv

```
# only if pip is not installed
sudo apt-get install python-pip python-dev build-essential
# upgrade pip, not necessary
sudo pip install --upgrade pip
# upgrade virtualenv
sudo pip install --upgrade virtualenv
```

1.2.2 Step 2: Get EvalAI code

If you haven't already created an ssh key and added it to your GitHub account, you should do that now by following these instructions.

• In your browser, visit https://github.com/Cloud-CV/EvalAI and click the `fork` button. You will need to be logged in to GitHub to do this.

• Open Terminal and clone your fork by

```
git clone git@github.com:YOUR_GITHUB_USER_NAME/EvalAI.git evalai
```

Don't forget to replace `YOUR_GITHUB_USER_NAME` with your git username.

1.2.3 Step 3: Setup codebase

• Create a python virtual environment and install python dependencies.

```
cd evalai
virtualenv venv
source venv/bin/activate
pip install -r requirements/dev.txt
```

• Rename `settings/dev.sample.py` as `dev.py`

```
cp settings/dev.sample.py settings/dev.py
```

• Create an empty postgres database and run database migration.

```
createdb evalai -U postgres
# update postgres user password
psql -U postgres -c "ALTER USER postgres PASSWORD 'postgres';"
# run migrations
python manage.py migrate
```

• For setting up frontend, please make sure that node(>=7.x.x), npm(>=5.x.x) and bower(>=1.8.x) are installed globally on your machine. Install npm and bower dependencies by running

```
npm install
bower install
```

1.2.4 Step 4: Start the development environment

- To run backend development server at http://127.0.0.1:8000, simply do:

```
# activate virtual environment if not activated
source venv/bin/activate
python manage.py runserver
```

- To run frontend development server at http://127.0.0.1:8888, simply do:

```
gulp dev:runserver
```

1.3 Fedora Installation Instructions

1.3.1 Step 1: Install prerequisites

- Install git

```
sudo yum install git-all
```

- Install postgres

```
sudo yum install postgresql postgresql-devel
```

If you still encounter issues with pg_config, you may need to add it to your PATH, i.e.

```
export PATH=$PATH:/usr/pgsql-x.x/bin
```

where `x.x` is your version, such as /usr/pgsql-9.5./bin.

- Install rabbitmq

```
# use the below commands to get Erlang on our system:
wget http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
wget http://rpms.famillecollet.com/enterprise/remi-release-6.rpm
sudo rpm -Uvh remi-release-6*.rpm epel-release-6*.rpm
# Finally, download and install Erlang:
sudo yum install -y erlang
# Once we have Erlang, we can continue with installing RabbitMQ:
wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.2.2/rabbitmq-server-3.2.2-1.
→noarch.rpm
rpm --import http://www.rabbitmq.com/rabbitmq-signing-key-public.asc
sudo yum install rabbitmq-server-3.2.2-1.noarch.rpm
```

- Install virtualenv

```
sudo yum -y install python-pip python-devel groupinstall 'Development Tools'
# upgrade pip, not necessary
sudo pip install --upgrade pip
# upgrade virtualenv
sudo pip install --upgrade virtualenv
```

1.3.2 Step 2: Get EvalAI code

If you haven't already created an ssh key and added it to your GitHub account, you should do that now by following these instructions.

- In your browser, visit https://github.com/Cloud-CV/EvalAI and click the `fork` button. You will need to be logged in to GitHub to do this.

- Open Terminal and clone your fork by

```
git clone git@github.com:YOUR_GITHUB_USER_NAME/EvalAI.git evalai
```

Don't forget to replace `YOUR_GITHUB_USER_NAME` with your git username.

1.3.3 Step 3: Setup codebase

- Create a python virtual environment and install python dependencies.

```
cd evalai
virtualenv venv
source venv/bin/activate
pip install -r requirements/dev.txt
```

- Rename `settings/dev.sample.py` as `dev.py`

```
cp settings/dev.sample.py settings/dev.py
```

- Create an empty postgres database and run database migration.

```
createdb evalai -U postgres
# update postgres user password
psql -U postgres -c "ALTER USER postgres PASSWORD 'postgres';"
# run migrations
python manage.py migrate
```

- For setting up frontend, please make sure that node(>=7.x.x), npm(>=5.x.x) and bower(>=1.8.x) are installed globally on your machine. Install npm and bower dependencies by running

```
npm install
bower install
```

1.3.4 Step 4: Start the development environment

- To run backend development server at backend

```
# activate virtual environment if not activated
source venv/bin/activate
python manage.py runserver
```

- To run frontend development server for at frontend

```
gulp dev:runserver
```

- To run backend development server at http://127.0.0.1:8000, simply do:

```
# activate virtual environment if not activated
source venv/bin/activate
python manage.py runserver
```

- To run frontend development server at http://127.0.0.1:8888, simply do:

```
gulp dev:runserver
```

1.3.5 Common Errors

Error: *You need to install postgresql-server-dev-X.Y for building a server-side extension or libpq-dev for building a client-side application.*

Solution: Install libpq-dev

```
sudo apt-get install libpq-dev
```

Possible solutions for the same problem can be found at link.

1.4 Windows Installation Instructions

Setting up EvalAI on your local machine is really easy. Follow this guide to setup your development machine.

1.4.1 Step 1: Install prerequisites

- Install Python 2.x, Git, PostgreSQL version >= 9.4, RabbitMQ and virtualenv, in your computer, if you don't have it already.

1.4.2 Step 2: Get EvalAI Code

- Get the source code on your machine via git.

```
git clone https://github.com/Cloud-CV/EvalAI.git evalai
```

1.4.3 Step 3: Setup the codebase

- Create a python virtual environment and install python dependencies.

```
cd evalai
virtualenv venv
cd venv/scripts
activate.bat      # run this command everytime before working on project
cd ../..
pip install -r requirements/dev.txt
```

- Rename `settings/dev.sample.py` as `dev.py` and change credential in `settings/dev.py`

```
cp settings/dev.sample.py settings/dev.py
```

Use your postgres username and password for fields `USER` and `PASSWORD` in `dev.py` file.

- Create an empty postgres database and run database migration. Make sure you have defined the PostgreSql path to the Environment Variables.

```
createdb evalai
```

Enter your password for authentication and a new database will be added.

```
python manage.py migrate
```

- Seed the database with some fake data to work with.

```
python manage.py seed
```

This command also creates a `superuser(admin)`, a `host user` and a `participant user` with following credentials.

SUPERUSER- username: `admin` password: `password`**HOST USER-** username: `host` password: `password`**PARTICIPANT USER-** username: `participant` password: `password`

1.4.4 Step 4: Start the development environment

- That's it. Now you can run development server at http://127.0.0.1:8000 (for serving backend)

```
python manage.py runserver
```

- Open a new cmd window with node>=(7.0.0) installed on your machine and type

```
npm install
```

- Install bower(1.8.0) globally by running:

```
npm install -g bower
```

- Now install the bower dependencies by running:

```
bower install
```

- If you running npm install behind a proxy server, use

```
npm config set proxy http://proxy:port
```

- Now to connect to dev server at http://127.0.0.1:8888 (for serving frontend)

```
gulp dev:runserver
```

- That's it, Open web browser and hit the url http://127.0.0.1:8888.

Creating a Challenge

One can create a challenge in EvalAI using either:

1. zip configuration

2. web interface

2.1 Challenge creation using zip configuration

2.1.1 Getting Started

Creating a challenge on EvalAI is a three-step process. You just need to upload the challenge details in a challenge configuration file (**YAML file**) and we will take care of the rest.

The challenge configuration file on EvalAI consists of following fields:

- **title**: Title of the challenge

- **short_description**: Short description of the challenge (preferably 140 characters max)

- **description**: Long description of the challenge (use a relative path for the html file, e.g. `challenge_details/description.html`)

- **evaluation_criteria**: Evaluation criteria and details of the challenge (use a relative path for the html file, e.g. `challenge_details/evaluation.html`)

- **terms_and_conditions**: Terms and conditions of the challenge (use a relative path for the html file, e.g. `challenge_details/tnc.html`)

- **image**: Logo of the challenge (use a relative path for the logo in the zip configuration, e.g. `images/logo/challenge_logo.jpg`). **Note**: The image must be in jpg, jpeg or png format.

- **submission_guidelines**: Submission guidelines of the challenge (use a relative path for the html file, e.g. `challenge_details/submission_guidelines.html`)

- **evaluation_script**: The evaluation script using which the submissions will be evaluated (path of the evaluation script file or folder relative to this YAML file.)

- **start_date**: Start DateTime of the challenge (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10) in `UTC` timezone

- **end_date**: End DateTime of the challenge (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10) in `UTC` timezone

- **published**: True/False (Boolean field that gives the flexibility to publish the challenge once approved by EvalAI Admin. Default is `False`)

- **allowed_email_domains**: A list of domains allowed to participate in the challenge. Leave blank if everyone is allowed to participate. (e.g. `["domain1.com", "domain2.org", "domain3.in"]` Participants with these email domains will only be allowed to participate.)

- **blocked_emails_domains**: A list of domains not allowed to participate in the challenge. Leave blank if everyone is allowed to participate. (e.g. `["domain1.com", "domain2.org", "domain3.in"]` The participants with these email domains will not be allowed to participate.)

- **leaderboard**:

  A leaderboard for a challenge on EvalAI consists of following subfields:

  - **id**: Unique integer field for each leaderboard entry

  - **schema**: Schema field contains the information about the rows of the leaderboard. Schema contains two keys in the leaderboard:

  1. `labels`: Labels are the header rows in the leaderboard according to which the challenge ranking is done.

  2. `default_order_by`: This key decides the default sorting of the leaderboard based on one of the labels defined above.

  The leaderboard schema for VQA Challenge 2017 looks something like this:

  ```
  {
      "labels": ["yes/no", "number", "others", "overall"],
      "default_order_by": "overall"
  }
  ```

  The above schema of leaderboard for VQA Challenge creates the leaderboard web interface like this:

VQA
Leaderboard

- **challenge_phases**:

  There can be multiple challenge phases in a challenge. A challenge phase in a challenge contains the following subfields:

  - **id**: Unique integer identifier for the challenge phase

  - **name**: Name of the challenge phase

  - **description**: Long description of the challenge phase (set relative path of the html file, e.g. `challenge_details/phase1_description.html`)

  - **leaderboard_public**: True/False (Boolean field that gives the flexibility to Challenge Hosts to make their leaderboard public or private. Default is `False`)

  - **is_public**: True/False (Boolean field that gives the flexibility to Challenge Hosts to either hide or show the challenge phase to participants. Default is `False`)

  - **leaderboard_public**: True/False (Boolean field that gives the flexibility to Challenge Hosts to either make the leaderboard public or private. Default is `False`)

  - **is_submission_public**: True/False (Boolean field that gives the flexibility to Challenge Hosts to either make the submissions by default public/private. Note that this will only work when the `leaderboard_public` property is set to true. Default is `False`)

  - **start_date**: Start DateTime of the challenge phase (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10)

  - **end_date**: End DateTime of the challenge phase (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10)

  - **test_annotation_file**: This file is used for ranking the submission made by a participant. An annotation file can be shared by more than one challenge phase. (Path of the test annotation file relative to this YAML file, e.g. `challenge_details/test_annotation.txt`)

2.1. Challenge creation using zip configuration   **11**

- **codename**: Challenge phase codename. Note that the codename of a challenge phase is used to map the results returned by the evaluation script to a particular challenge phase. The codename specified here should match with the codename specified in the evaluation script to perfect mapping.

- **max_submissions_per_day**: Positive integer which tells the maximum number of submissions per day to a challenge phase.

- **max_submissions**: Positive integer that decides the overall maximum number of submissions that can be done to a challenge phase.

- **dataset_splits**:

  A dataset in EvalAI is the main entity on which an AI challenge is based on. Participants are expected to make submissions corresponding to different splits of the corresponding dataset. A dataset is generally divided into different parts and each part is called dataset split. Generally, a dataset has three different splits:

  1. Training set

  2. Validation set

  3. Test set

  - **id**: Unique integer identifier for the dataset split

  - **name**: Name of the dataset split (it must be unique for every dataset split)

  - **codename**: Codename of dataset split. Note that the codename of a dataset split is used to map the results returned by the evaluation script to a particular dataset split in EvalAI's database. Please make sure that no two dataset splits have the same codename. Again, make sure that the dataset split's codename match with what is in the evaluation script provided by the challenge host.

- **challenge_phase_splits**:

  A challenge phase split is a relation between a challenge phase and dataset splits for a challenge (many to many relation). This is used to set the privacy of submissions (public/private) to different dataset splits for different challenge phases.

  - **challenge_phase_id**: Id of challenge_phase (Gets the challenge phase details to map with)

  - **leaderboard_id**: Id of leaderboard (Given above)

  - **dataset_split_id**: Id of dataset split (Given above)

  - **visibility**: Enter any of the positive integers given below.

    a) HOST -> 1

    b) OWNER AND HOST -> 2

    c) PUBLIC -> 3

2.1.2 Sample zip configuration file

Here is a sample configuration file for a challenge with 1 phase and 2 dataset split:

```
title: Challenge Title
short_description: Short description of the challenge (preferably 140 characters)
description: description.html
evaluation_details: evaluation_details.html
terms_and_conditions: terms_and_conditions.html
image : logo.jpg
submission_guidelines: submission_guidelines.html
evaluation_script: evaluation_script.zip
```

(continues on next page)

```yaml
start_date: 2017-06-09 20:00:00
end_date: 2017-06-19 20:00:00
published: True

leaderboard:
  - id: 1
    schema: {"labels": ["yes/no", "number", "others", "overall"], "default_order_by":
→"overall"}
  - id: 2
    schema: {"labels": ["yes/no", "number", "others", "overall"], "default_order_by":
→"yes/no"}

challenge_phases:
  - id: 1
    name: Challenge name of the challenge phase
    description: challenge_phase_description.html
    leaderboard_public: True
    is_public: True
    start_date: 2017-06-09 20:00:00
    end_date: 2017-06-19 20:00:00
    test_annotation_file: test_annotation.txt
    codename: Challenge phase codename
    max_submissions_per_day: 100
    max_submissions: 1000

dataset_splits:
  - id: 1
    name: Name of the dataset split
    codename: codename of dataset split 1
  - id: 2
    name: Name of the dataset split
    codename: codename of dataset split 2

challenge_phase_splits:
  - challenge_phase_id: 1
    leaderboard_id: 2
    dataset_split_id: 1
    visibility: 3
  - challenge_phase_id: 1
    leaderboard_id: 1
    dataset_split_id: 2
    visibility: 3
```

2.1.3 Challenge Creation Examples

Please see this repository for examples on the different types of challenges on EvalAI.

2.1.4 Next Steps

The next step is to create a zip file that contains the YAML config file, all the HTML templates for the challenge description, challenge phase description, evaluation criteria, submission guidelines, evaluation script, test annotation file(s) and challenge logo (optional).

The final step is to create a challenge host team for the challenge on EvalAI. After that, just upload the zip folder created in the above steps and the challenge will be created.

If you have issues in creating a challenge on EvalAI, please feel free to create an issue on our Github Issues Page.

2.2 Create challenge using web interface

Todo: We are working on this feature and will keep you updated.

Writing Evaluation Script

Each challenge has an evaluation script, which evaluates the submission of participants and returns the scores which will populate the leaderboard.

The logic for evaluating and judging a submission is customizable and varies from challenge to challenge, but the overall structure of evaluation scripts are fixed due to architectural reasons.

Evaluation scripts are required to have an `evaluate` function. This is the main function, which is used by workers to evaluate the submission messages.

The syntax of evaluate function is:

```python
def evaluate(test_annotation_file, user_annotation_file, phase_codename, **kwargs):

    pass
```

It receives three arguments, namely:

- test_annotation_file

This is the path to the annotation file for the challenge. This is the file uploaded by the Challenge Host while creating a Challenge.

- user_annotation_file

This is the path of the file submitted by the user for a particular phase.

- phase_codename

This is the `ChallengePhase` model codename. This is passed as an argument, so that the script can take actions according to the phase.

After reading the files, some custom actions can be performed. This varies per challenge.

The `evaluate()` method also accepts keyword arguments. By default, we provide you metadata of each submission to your challenge which you can use to send notifications to your slack channel or to some other webhook service. Following is an example code showing how to get the submission metadata in your evaluation script and send a slack notification if the accuracy is more than some value `X` (X being 90 in the expample given below).

```python
def evaluate(test_annotation_file, user_annotation_file, phase_codename, **kwargs):

    submission_metadata = kwargs.get("submission_metadata")
    print submission_metadata

    # Do stuff here
    # Let's set `score` to 90 as an example

    score = 91
    if score > 90:
        slack_data = kwargs.get("submission_metadata")
        webhook_url = "Your Slack Webhook URL comes here"
        # To know more about slack webhook, checkout this link: https://api.slack.com/
→incoming-webhooks

        response = requests.post(
            webhook_url,
            data=json.dumps({'text': "*Flag raised for submission:* \n \n" +
→str(slack_data)}),
            headers={'Content-Type': 'application/json'})

    # Do more stuff here
```

The above example can be modified and used to find if some participant team is cheating or not. There are many more ways for which you can use this metadata.

After all the processing is done, this script will send an output, which is used to populate the leaderboard. The output should be in the following format:

```python
output = {}
output['result'] = [
    {
        'dataset_split_1': {
            'score': score,
        }
    },
    {
        'dataset_split_2': {
            'score': score,
        }
    }
]
return output
```

`output` should contain a key named `result`, which is a list containing entries per dataset split that is available for the challenge phase under consideration (In the function definition of `evaluate` shown above, the argument: `phase_codename` will receive the *codename* for the challenge phase against which the submission was made). Each entry in the list should be an object that has a key with the corresponding dataset split codename(`dataset_split_1` and `dataset_split_2` for this example). Each of these dataset split objects contains various keys (`score` in this example), which are then displayed as columns in the leaderboard.

> NOTE: `dataset_split_1` and `dataset_split_2` are codenames for dataset splits that should be evaluated with each submission for the challenge phase obtained via *phase_codename*.

Note: If your evaluation script uses some precompiled libraries (MSCOCO for example), then make sure that the library is compiled against a Linux Distro (Ubuntu 14.04 recommended). Libraries compiled against OSx or Windows might or might not work properly.

Submission

4.1 How is a submission processed?

We are using REST APIs along with Queue based architecture to process submissions. When a participant makes a submission for a challenge, a REST API with url pattern `jobs:challenge_submission` is called. This API does the task of creating a new entry for submission model and then publishes a message to exchange `evalai_submissions` with a routing key of `submission.*.*`.

```
 User makes    --> API  --> Publish  --> RabbitMQ  --> Queue  --> Submission
a submission                message     Exchange                   worker(s)
```

Exchange receives the message and then routes it to the queue `submission_task_queue`. At the end of `submission_task_queue` are workers (scripts/workers/submission_worker.py) which processes the submission message.

The worker can be run with

```
# assuming the current working directory is where manage.py lives
python scripts/workers/submission_worker.py
```

4.2 How does submission worker function?

Submission worker is a python script which mostly runs as a daemon on a production server and simply acts as a python process in a development environment. To run submission worker in a development environment:

```
python scripts/workers/submission_worker.py
```

Before a worker fully starts, it does the following actions:

- Creates a new temporary directory for storing all its data files.

- Fetches the list of active challenges from the database. Active challenges are published challenges whose start date is less than present time and end date greater than present time. It loads all the challenge evaluation scripts in a variable called `EVALUATION_SCRIPTS`, with the challenge id as its key. The maps looks like this:

```
EVALUATION_SCRIPTS = {
    <challenge_pk> : <evalutaion_script_loaded_as_module>,
    ....
}
```

- Creates a connection with RabbitMQ by using the connection parameters specified in `settings.RABBITMQ_PARAMETERS`.

- After the connection is successfully created, creates an exchange with the name `evalai_submissions` and two queues, one for processing submission message namely `submission_task_queue`, and other for getting add challenge message.

- `submission_task_queue` is then bound with the routing key of `submission.*.*` and add challenge message queue is bound with a key of `challenge.add.*` Whenever a queue is bound to a exchange with any key, it will route the message to the corresponding queue as soon as the exchange receives a message with a key.

- Binding to any queue is also accompanied with a callback which basically takes a function as an argument. This function specifies what should be done when the queue receives a message.

e.g. `submission_task_queue` is using `process_submission_callback` as a function, which means that when a message is received in the queue, `process_submission_callback` will be called with the message passed as an argument.

Expressing it informally it will be something like

> *Queue*: Hey *Exchange*, I am `submission_task_queue`. I will be listening to messages from you on binding key of `submission.*.*`
>
> *Exchange*: Hey *Queue*, Sure! When I receive a message with a routing key of `submission.*.*`, I will give it to you
>
> *Queue*: Thanks a lot.
>
> *Queue*: Hey *Worker*, Just for the record, when I receive a new message for submission, I want `process_submission_callback` to be called. Can you please make a note of it?
>
> *Worker*: Sure *Queue*, I will invoke `process_submission_callback` whenever you receive a new message.

When a worker starts, it fetches active challenges from the database and then loads all the challenge evaluation scripts in a variable called `EVALUATION_SCRIPTS`, with challenge id as its key. The map would look like

```
EVALUATION_SCRIPTS = {
    <challenge_pk> : <evalutaion_script_loaded_as_module>,
    ....
}
```

After the challenges are successfully loaded, it creates a connection with the RabbitMQ Exchange `evalai_submissions` and then listens on the queue `submission_task_queue` with a binding key of `submission.*.*`.

4.3 How is submission made?

When the user makes a submission on the frontend, the following actions happen sequentially

- As soon as the user submits a submission, a REST API with the URL pattern `jobs:challenge_submission` is called.

- This API fetches the challenge and its corresponding challenge phase.

- This API then checks if the challenge is active and challenge phase is public.

- It fetches the participant team's ID and its corresponding object.

- After all these checks are complete, a submission object is saved. The saved submission object includes **participant team id** and **challenge phase id** and **username** of the participant creating it.

- At the end, a submission message is published to exchange `evalai_submissions` with a routing key of `submission.*.*`.

4.4 Format of submission messages

The format of the message is

```
{
    "challenge_id": <challenge_pk_here>,
    "phase_id": <challenge_phase_pk_here>,
    "submission_id": <submission_pk_here>
}
```

This message is published with a routing key of `submission.*.*`

4.5 How workers process submission message

Upon receiving a message from `submission_task_queue` with a binding key of `submission.*.*`, `process_submission_callback` is called. This function does the following:

- It fetches the challenge phase and submission object from the database using the challenge phase id and submission id received in the message.

- It then downloads the required files like input_file, etc. for submission in its computation directory.

- After this, the submission is run. Submission is initially marked in **RUNNING** state. The `evaluate` function of `EVALUATION_SCRIPTS` map with key of the challenge id is called. The `evaluate` function takes in the annotation file path, the user annotation file path, and the challenge phase's code name as arguments. Running a submission involves temporarily updating `stderr` and `stdout` to different locations other than standard locations. This is done so as to capture the output and any errors produced when running the submission.

- The output from the `evaluate` function is stored in a variable called `submission_output`. Currently, the only way to check for the occurrence of an error is to check if the key `result` exists in `submission_output`.

  - If the key does not exist, then the submission is marked as **FAILED**.

  - If the key exists, then the variable `submission_output` is parsed and `DataSetSplit` objects are created. LeaderBoardData objects are also created (in bulk) with the required parameters. Finally, the submission is marked as **FINISHED**.

- The value in the temporarily updated `stderr` and `stdout` are stored in files named `stderr.txt` and `stdout.txt` which are then stored in the submission instance.

- Finally, the temporary computation directory allocated for this submission is removed.

4.6 Notes

- REST API with url pattern `jobs:challenge_submission`. Here *jobs* is application namespace and *challenge_submission* is instance namespace. You can read more about url namespace

Architecture

EvalAI helps researchers, students, and data scientists to create, collaborate, and participate in various AI challenges organized around the globe. To achieve this, we leverage some of the best open source tools and technologies.

5.1 Technologies that the project uses:

5.1.1 Django

Django is the heart of the application, which powers our backend. We use Django version 1.10.

5.1.2 Django Rest Framework

We use Django Rest Framework for writing and providing REST APIs. Its permission and serializers have helped write a maintainable codebase.

5.1.3 RabbitMQ

We currently use RabbitMQ for queueing submission messages which are then later on processed by a Python worker.

5.1.4 PostgreSQL

PostgreSQL is used as our primary datastore. All our tables currently reside in a single database named `evalai`

5.1.5 Angular JS

Angular JS is a well-known framework that powers our frontend.

Architectural Decisions

This is a collection of records for architecturally significant decisions.

6.1 URL Patterns

We follow a very basic, yet strong convention for URLs, so that our rest APIs are properly namespaced. First of all, we rely heavily on HTTP verbs to perform **CRUD** actions.

For example, to perform **CRUD** operation on *Challenge Host Model*, the following URL patterns will be used.

- `GET /hosts/challenge_host_team` - Retrieves a list of challenge host teams

- `POST /hosts/challenge_host_team` - Creates a new challenge host team

- `GET /hosts/challenge_host_team/<challenge_host_team_id>` - Retrieves a specific challenge host team

- `PUT /hosts/challenge_host_team/<challenge_host_team_id>` - Updates a specific challenge host team

- `PATCH /hosts/challenge_host_team/<challenge_host_team_id>` - Partially updates a specific challenge host team

- `DELETE /hosts/challenge_host_team/<challenge_host_team_id>` - Deletes a specific challenge host team

Also, we have namespaced the URL patterns on a per-app basis, so URLs for *Challenge Host Model*, which is in the *hosts* app, will be

```
/hosts/challenge_host_team
```

This way, one can easily identify where a particular API is located.

We use underscore _ in URL patterns.

6.2 Processing submission messages asynchronously

When a submission message is made, a REST API is called which saves the data related to the submission in the database. A submission involves the processing and evaluation of `input_file`. This file is used to evaluate the submission and then decide the status of the submission, whether it is *FINISHED* or *FAILED*.

One way to process the submission is to evaluate it as soon as it is made, hence blocking the participant's request. Blocking the request here means to send the response to the participant only when the submission has been made and its output is known. This would work fine if the number of the submissions made is very low, but this is not the case.

Hence we decided to process and evaluate submission message in an asynchronous manner. To process the messages this way, we need to change our architecture a bit and add a Message Framework, along with a worker so that it can process the message.

Out of all the awesome messaging frameworks available, we have chosen RabbitMQ because of its transactional nature and reliability. Also, RabbitMQ is easily horizontally scalable, which means we can easily handle the heavy load by simply adding more nodes to the cluster.

For the worker, we went ahead with a normal python worker, which simply runs a process and loads all the required data in its memory. As soon as the worker starts, it listens on a RabbitMQ queue named `submission_task_queue` for new submission messages.

6.3 Submission Worker

The submission worker are responsible for processing submission messages. It listens on a queue named `submission_task_queue`, and on receiving a message for a submission, it processes and evaluates the submission.

One of the major design changes that we decided to implement in the submission worker was to load all the data related to the challenge in the worker's memory, instead of fetching it every time a new submission message arrives. So the worker, when starting, fetches the list of active challenges from the database and then loads it into memory by maintaining the map `EVALUATION_SCRIPTS` on challenge id. This was actually a major performance improvement.

Another major design change that we incorporated here was to dynamically import the challenge module and to load it in the map instead of invoking a new python process every time a submission message arrives. So now whenever a new message for a submission is received, we already have its corresponding challenge module being loaded in a map called `EVALUATION_SCRIPTS`, and we just need to call

```
EVALUATION_SCRIPTS[challenge_id].evaluate(*params)
```

This was again a major performance improvement, which saved us from the task of invoking and managing Python processes to evaluate submission messages. Also, invoking a new python process every time for a new submission would have been really slow.

Directory Structure

7.1 Django Apps

EvalAI is a Django-based application, hence it leverages the concept of Django apps to properly namespace the functionalities. All the apps can be found in the `apps` directory situated in the root folder.

Some important apps along with their main uses are:

- **Challenges**

This app handles all the workflow related to creating, modifying, and deleting challenges.

- **Hosts**

This app is responsible for providing functionalities to the challenge hosts/organizers.

- **Participants**

This app serves users who want to take part in any challenge. It contains code for creating a Participant Team, through which they can participate in any challenge.

- **Jobs**

One of the most important apps, responsible for processing and evaluating submissions made by participants. It contains code for creating a submission, changing the visibility of the submission and populating the leaderboard for any challenge.

- **Web**

This app serves some basic functionalities like providing support for contact us or adding a new contributor to the team, etc.

- **Accounts**

As the name indicates, this app deals with storing and managing data related to user accounts.

- **Base**

A placeholder app which contains the code that is used across various other apps.

7.2 Settings

Settings are used across the backend codebase by Django to provide config values on a per-environment basis. Currently, the following settings are available:

- **dev**

Used in development environment

- **testing**

Used whenever test cases are run

- **staging**

Used on staging server

- **production**

Used on production server

7.3 URLs

The base URLs for the project are present in `evalai/urls.py`. This file includes URLs of various applications, which are also namespaced by the app name. So URLs for the `challenges` app will have its app namespace in the URL as `challenges`. This actually helps us separate our API based on the app.

7.4 Frontend

The whole codebase for the frontend resides in a folder named `frontend` in the root directory

7.5 Scripts

Scripts contain various helper scripts, utilities, python workers. It contains the following folders:

- **migration**

Contains some of the scripts which are used for one-time migration or formatting of data.

- **tools**

A folder for storing helper scripts, e.g. a script to fetch pull request

- **workers**

One of the main directories, which contains the code for submission worker. Submission worker is a normal python worker which is responsible for processing and evaluating submission of a user. The command to start a worker is:

```
python scripts/workers/submission_worker.py
```

7.6 Test Suite

All of the codebase related to testing resides in `tests` folder in the root directory. In this directory, tests are namespaced according to the app, e.g. tests for `challenges` app lives in a folder named `challenges`.

7.7 Management Commands

To perform certain actions like seeding the database, we use Django management commands. Since the management commands are common throughout the project, they are present in `base` application directory. At the moment, the only management command is `seed`, which is used to populate the database with some random values. The command can be invoked by calling

```
python manage.py seed
```

Tutorial to participate in a challenge in EvalAI

Participating in EvalAI is really easy. One just needs to create an account and a participant team in order to participate in a challenge.

If you are already familiar with the flow of EvalAI, you may want to skip this section else please follow the following steps to participate in a challenge (VQA Challenge 2017 in this example):

8.1 1. Visit evalai.cloudcv.org

Open EvalAI website.

8.2 2. Sign up or Log in

Sign Up and fill in your credentials or log in if you have already registered.

After signing up you would be on the dashboard page.

8.3 4. Choose challenge

Then, go to challenges section and choose an active challenge.

8.4 5. Challenge Page

After reading the challenge instructions on the challenge page, you can participate in the challenge.

8.5 6. Create Participant Team

Create a participant team if there isn't any or you can select from the existing ones.

Click on 'Participate' tab after selecting a team.

Tada! you have successfully participated in a challenge.

Frequently Asked Questions

9.1 Q. How to start contributing?

EvalAI's issue tracker is good place to start. If you find something that interests you, comment on the thread and we'll help get you started. Alternatively, if you come across a new bug on the site, please file a new issue and comment if you would like to be assigned. Existing issues are tagged with one or more labels, based on the part of the website it touches, its importance etc., which can help you select one.

9.2 Q. What are the technologies that EvalAI uses?

9.2.1 Django

Django is the heart of the application, which powers our backend. We use Django version 1.10.

9.2.2 Django Rest Framework

We use Django Rest Framework for writing and providing REST APIs. It's permission and serializers have helped write a maintainable codebase.

9.2.3 RabbitMQ

We currently use RabbitMQ for queueing submission messages which are then later on processed by a Python worker.

9.2.4 PostgreSQL

PostgresSQL is used as our primary datastore. All our tables currently reside in a single database named evalai.

9.2.5 Angular JS - ^1.6.1

Angular JS is a well-known framework that powers our frontend.

9.3 Q. Where could I learn Github Commands?

Refer to Github Guide.

9.4 Q. Where could I learn Markdown?

Refer to MarkDown Guide.

9.5 Q. What to do when coverage decreases in your pull request?

Coverage decreases when the existing test cases don't test the new code you wrote. If you click coverage, you can see exactly which all parts aren't covered and you can write new tests to test the parts.

9.6 Common Errors during installation

9.6.1 Q. While using `pip install -r dev/requirement.txt`

```
Writing manifest file 'pip-egg-info/psycopg2.egg-info/SOURCES.txt'
Error: You need to install postgresql-server-dev-X.Y for building a server-side
→extension or
libpq-dev for building a client-side application.
---------------------------------------
Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-build-qIjU8G/
→psycopg2/
```

Use the following commands in order to solve the error:

1. `sudo apt-get install postgresql`
2. `sudo apt-get install python-psycopg2`
3. `sudo apt-get install libpq-dev`

9.6.2 Q. While using `pip install -r dev/requirement.txt`

```
Command "python setup.py egg_info" failed with error code 1 in
/private/var/folders/c7/b45s17816zn_b1dh3g7yzxrm0000gn/T/pip-build- GM2AG/psycopg2/
```

Firstly check that you have installed all the mentioned dependencies. Then, Upgrade the version of postgresql to 10.1 in order to solve it.

9.6.3 Q. Getting an import error

```
Couldn't import Django,"when using command python manage.py migrate
```

Firstly, check that you have activated the virtualenv. Install python dependencies using the following commands on the command line

```
cd evalai
pip install -r requirements/dev.txt
```

9.6.4 Q. Getting Mocha Error

```
Can not load reporter "mocha",it is not registered
```

Uninstall karma and then install

```
npm uninstall -g generator-karma && npm install -g generator-angular.
```

9.6.5 Q. While trying to execute `bower install`

```
bower: command not found
```

Execute the following command first :

```
npm install -g bower
```

9.6.6 Q. While trying to execute `gulp dev:runserver`

```
gulp: command not found
```

Execute the following command first

```
npm install -g gulp-cli
```

9.6.7 Q. While executing `gulp dev:runserver`

```
events.js:160
throw er; // Unhandled 'error' event
^
Error: Gem sass is not installed.
```

Execute the following command first :

```
gem install sass
```

9.6.8 Q. While trying to install `npm config set proxy http://proxy:port` on UBUNTU, I get the following error:

```
ubuntu@ubuntu-Inspiron-3521:~/Desktop/Python-2.7.14$ npm install -g angular-cli
npm ERR! Linux 4.4.0-21-generic
npm ERR! argv "/usr/bin/nodejs" "/usr/bin/npm" "install" "-g" "angular-cli"
npm ERR! node v4.2.6
npm ERR! npm  v3.5.2
npm ERR! code ECONNRESET

npm ERR! network tunneling socket could not be established, cause=getaddrinfo␣
→ENOTFOUND proxy proxy:80
npm ERR! network This is most likely not a problem with npm itself
npm ERR! network and is related to network connectivity.
npm ERR! network In most cases you are behind a proxy or have bad network settings.
npm ERR! network
npm ERR! network If you are behind a proxy, please make sure that the
npm ERR! network 'proxy' config is set properly.  See: 'npm help config'

npm ERR! Please include the following file with any support request:
npm ERR!     /home/ubuntu/Desktop/Python-2.7.14/npm-debug.log
```

To solve, execute the following commands:

1. npm config set registry=registry.npmjs.org

If the above does not work, try deleting them by following commands:

1. npm config delete proxy

2. npm config delete https-proxy

Then, start the instllation process of frontend once more.

9.6.9 Q. While using docker, I am getting the following error on URL http://localhost:8888/

```
Cannot Get \
```

Try removing the docker containers and then building them again.

9.6.10 Q. Getting the following error while running `python manage.py seed`

```
Starting the database seeder. Hang on... Exception while running run() in 'scripts.
→seed' Database successfully seeded
```

Change the pyhton version to 2.7.x . The problem might be because of the pyhton 3.0 version.

9.6.11 Q. Getting the following error while executing command `createdb evalai -U postgres`

```
createdb: could not connect to database template1: FATAL: Peer authentication failed␣
→for user "postgres"
```

Try creating a new user and then grant all the privileges to it and then create a db.

9.6.12 Q. Getting the following error while executing `npm install`

```
npm WARN generator-angular@0.16.0 requires a peer of generator-
karma@>=0.9.0 but none was installed.
```

Uninstall and then install karma again and also don't forget to clean the global as well as project npm cache. Then try again the step 8.

Migrations

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into. - Django Migration Docs

10.1 Creating a migration

- We recommend you to create migrations per app, where the changes are only about a single issue or feature.

```
# migration only for jobs app
python manage.py makemigrations jobs
```

- Always create named migrations. You can name migrations by passing -n or --name argument

```
python manage.py makemigrations jobs -n=execution_time_limit
# or
python manage.py makemigrations jobs --name=execution_time_limit
```

- While creating migrations on local environment, don't forget to add development settings.

```
python manage.py makemigrations
```

The following is an example of a complete named migration for the `jobs` app, wherein a execution time limit field is added to the `Submission` model:

```
python manage.py makemigrations jobs --name=execution_time_limit
```

- Files create after running `makemigrations` should be committed along with other files

- While creating a migration for your concerned change, it may happen that some other changes are also there in the migration file. Like adding a `execution_time_limit` field on `Submission` model also brings in the change for `when_made_public` being added. In that case, open an new issue and clearly mention the

issue over there. If possible fix the issue yourself, by opening a new branch and creating migrations only for the concerned part. The idea here is that a commit should only include its concerned migration changes and nothing else.

Contributing guidelines

Thank you for your interest in contributing to EvalAI! Here are a few pointers on how you can help.

11.1 Setting things up

To set up the development environment, follow the instructions in our README.

11.2 Finding something to work on

EvalAI's issue tracker is good place to start. If you find something that interests you, comment on the thread and we'll help get you started.

Alternatively, if you come across a new bug on the site, please file a new issue and comment if you would like to be assigned. Existing issues are tagged with one or more labels, based on the part of the website it touches, its importance etc., which can help you select one.

If neither of these seem appealing, please post on our channel and we will help find you something else to work on.

11.3 Instructions to submit code

Before you submit code, please talk to us via the issue tracker so we know you are working on it.

Our central development branch is *development*. Coding is done on feature branches based off of development and merged into it once stable and reviewed. To submit code, follow these steps:

1. Create a new branch off of development. Select a descriptive branch name. We highly encourage you to use *autopep8* to follow the PEP8 styling. Run the following command before creating the pull request:

```
autopep8 --in-place --exclude venv,docs --recursive .

git fetch upstream
git checkout master
git merge upstream/master
git checkout -b your-branch-name
```

2. Commit and push code to your branch:

   - Commits should be self-contained and contain a descriptive commit message.

   - Please make sure your code is well-formatted and adheres to PEP8 conventions (for Python) and the airbnb style guide (for JavaScript). For others (Lua, prototxt etc.) please ensure that the code is well-formatted and the style consistent.

   - Please ensure that your code is well tested.

   ```
   git commit -a -m "{{commit_message}}"
   git push origin {{branch_name}}
   ```

3. Once the code is pushed, create a pull request:

   - On your Github fork, select your branch and click "New pull request". Select "master" as the base branch and your branch in the "compare" dropdown. If the code is mergeable (you get a message saying "Able to merge"), go ahead and create the pull request.

   - Check back after some time to see if the Travis checks have passed, if not you should click on "Details" link on your PR thread at the right of "The Travis CI build failed", which will take you to the dashboard for your PR. You will see what failed / stalled, and will need to resolve them.

   - If your checks have passed, your PR will be assigned a reviewer who will review your code and provide comments. Please address each review comment by pushing new commits to the same branch (the PR will automatically update, so you don't need to submit a new one). Once you are done, comment below each review comment marking it as "Done". Feel free to use the thread to have a discussion about comments that you don't understand completely or don't agree with.

   - Once all comments are addressed, the reviewer will give an LGTM ('looks good to me') and merge the PR.

Congratulations, you have successfully contributed to Project EvalAI!

Pull Request

Contributing to EvalAI is really easy. Just follow these steps to get started.

Step 1: Fork

1. Fork the EvalAI repository from the repository.

Step 2: Selecting an issue

1. Select a suitable issue that will be easy for you to fix. Moreover, you can also take the issues based on their labels. All the issues are labelled according to its difficulty.

2. After selecting an issue, ask the maintainers of the project to assign it to you and they will assign it based on its availability.

3. Once it gets assigned, create a branch from your fork's updated master branch using the following command: `git checkout -b branch_name`

4. Start working on the issue.

Step 3: Committing Your Changes

1. After making the changes, you need to add your files to your local git repository.

2. To add your files, use the following commands:

- To add only modified files, use `git add -u`

- To add a new file, use `git add file_path_from_local_git_repository`

- To add all files, use `git add .`

1. Once you have added your files, you need to commit your changes. Always **create a very meaningful commit message related to the changes that you have done. Try to write the commit message in present imperative tense. Also namespace the commit message so that it becomes self-explanatory by just looking at the commit message.** For example,

   Docs: Add verbose setup docs for ubuntu

Step 4: Creating a Pull Request

1. Before creating a Pull Request, you need to first rebase your branch with the upstream master.

2. To rebase your branch, use following commands: `git fetch upstream git rebase upstream/master`

3. After rebasing, push the changes to your forked repository. `git push origin branch_name`

4. After pushing the code, create a Pull Request.

5. When creating a pull request, be sure to add a comment including these keywords, and also mention any maintainer to reviewing it.

Note:

- If you have any doubts, don't hestitate to ask in the comments. You may also add in any relevant content.

- After the maintainers review your changes, fix the code as suggested. Don't forget to add, commit, and push your code to the same branch.

Once you have completed the above steps, you have successfully created a Pull Request to EvalAI.

Glossary

13.1 Challenge

An event, run by an institute or organization, wherein a number of researchers, students, and data scientists participate and compete with each other over a period of time. Each challenge has a start time and generally an end time too.

13.2 Challenge Host

A member of the host team who organizes a challenge. In our system, it is a form of representing a user. This user can be in the organizing team of many challenges, and hence for each challenge, its challenge host will be different.

13.3 Challenge Host Team

A group of challenge hosts who organizes a challenge. They are identified by a unique team name.

13.4 Challenge Phase Split

A challenge phase split is the relation between a challenge phase and dataset splits for a challenge with a many-to-many relation. This is used to set the privacy of submissions (public/private) to different dataset splits for different challenge phases.

13.5 Dataset

A dataset in EvalAI is the main entity in which an AI challenge is based on. Participants are expected to make submissions corresponding to different splits of the corresponding dataset.

13.6 Dataset Split

A dataset is generally divided into different parts called dataset split. Generally, a dataset has three different splits:

- Training set
- Validation set
- Test set

13.7 EvalAI

EvalAI is an open-source web platform that aims to be the state of the art in AI. Its goal is to help AI researchers, practitioners, and students to host, collaborate, and participate in AI challenges organized around the globe.

13.8 Leaderboard

The leaderboard can be defined as a scoreboard listing the names of the teams along with their current scores. Currently, each challenge has its own leaderboard.

13.9 Phase

A challenge can be divided into many phases (or challenge phases). A challenge phase can have the same or different start and end date than the challenge start and end date.

13.10 Participant

A member of the team competing against other teams for any particular challenge. It is a form of representing a user. A user can participate in many challenges, hence for each challenge, its participant entry will be different.

13.11 Participant Team

A group of one or more participants who are taking part in a challenge. They are identified uniquely by a team name.

13.12 Submission

A way of submitting your results to the platform, so that it can be evaluated and ranked amongst others. A submission can be public or private, depending on the challenge.

13.13 Submission Worker

A python script which processes submission messages received from a queue. It does the heavy lifting task of receiving a submission, performing mandatory checks, and then evaluating the submission and updating its status in the database.

13.14 Team

A model, present in `web` app, which helps CloudCV register new contributors as a core team member or simply an open source contributor.

13.15 Test Annotation File

This is generally a file uploaded by a challenge host and is associated with a challenge phase. This file is used for ranking the submission made by a participant. An annotation file can be shared by more than one challenge phase. In the codebase, this is present as a file field attached to challenge phase model.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search