#Chitter

social messaging app with oAuth2 authentication service

1

oAuth v2 Authentication Server

2

Twitter style messaging API

3

Authorisation server

#Chitter

social messaging app with oAuth2 authentication service

Deployed in Amazon AWS cloud as distributed service for high availability and fault tolerance

Public API endpoints:

http://13.40.180.161:8000/o/

http://13.40.180.161:8000/authentication/

http://13.40.180.161:8000/admin/

http://13.40.180.161:8000/v1/ {message|feedback|topic}

#Chitter

social messaging app with oAuth2 authentication service

At the core of the application we have an authorisation server (running as an API service)

Hosts the protected user accounts.

The authorization server verifies the identity of the user then issues access tokens to the application.

Endpoints:

http://13.40.180.161:8000/authentication/

{register | token | token/refresh | token/revoke}

#Chitter

social messaging app with oAuth2 authentication service

Chitter app is deployed as a service that allows people to share or “tweet” messages.

These messages can be discovered by other users based on usernames or by topics or "hashtags".

Users interacting with messages have option to either like a message, dislike a message or comment on it.

All messages have expiry date after which reactions are not allowed.

Endpoints: http://13.40.180.161:8000/v1/ {message| feedback| topic}

#Chitter

social messaging app with oAuth2 authentication service

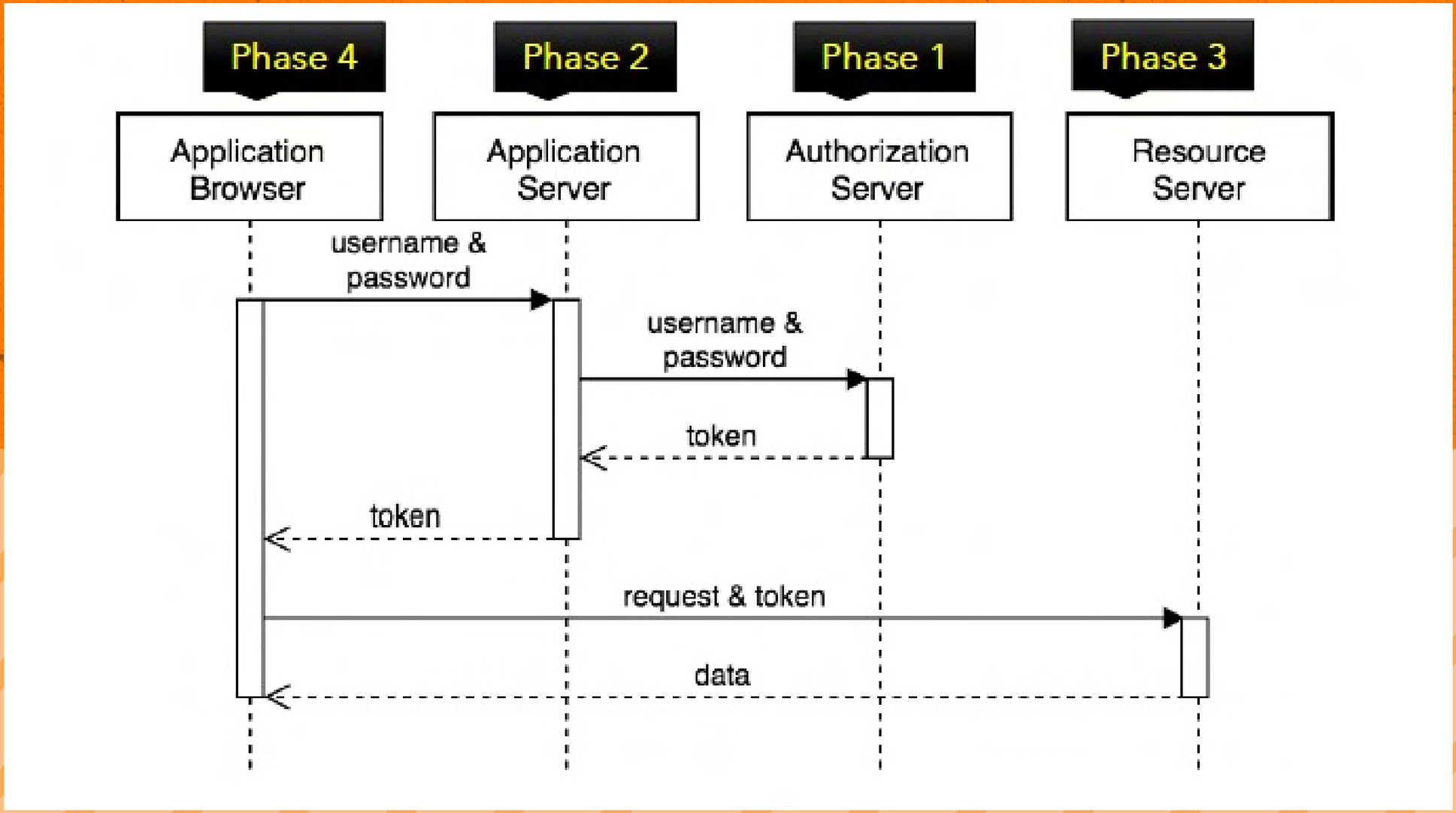
We have built a ‘client application’ to interact with our cloud based application using python.

All data sent / received by the API is in JSON format .


```
[
    {
        "post_identifier": 70,
        "topic": [
            {
                "id": 5,
                "topic_name": "T"
            }
        ],
        "title": "Chitter app",
        "message": "This is an amazing social messaging application",
        "creation_timestamp": "2023-04-09T03:53:24Z",
        "expiry_in_seconds": "2023-04-09T05:54:01Z",
        "username": "admin",
        "likes": 3,
        "dislikes": 0,
        "total_interactions": 0,
        "feedbacks": [],
        "live_status": true,
        "live_time_remaining": "7123.468853"
    }
]
```

```
[
    "title": "Chitter app",
    "message": "This is an amazing social messaging application",
    "creation_timestamp": "2023-04-09T03:53:24Z",
    "expiry_in_seconds": "2023-04-09T05:54:01Z",
    "username": "admin",
    "likes": 3,
    "dislikes": 0,
    "total_interactions": 0,
    "feedbacks": [],
    "live_status": true,
    "live_time_remaining": "7123.468853"
  }
]
```

How have we implemented oAuth v2 authentication?



Data Modeling

Database tables are written as python classes

```python
class Messages(models.Model):
    #This field acts as primary key and identifies a message.
    post_identifier = models.AutoField(primary_key=True)
    #This many to many relation field allows user to choose more than one topics for a message
    topic = models.ManyToManyField('Topics')
    #This field is used to store 'title' of a Message
    title = models.CharField(max_length=100)
    #This field is used to store message.
    message = models.TextField()
    #Creation timestamp is set to current date/time by default. No need to send this object in
    creation_timestamp = models.DateTimeField(default=timezone.now)
```

```python
class Feedback(models.Model):
    #If user likes a post, turn this field to true
    is_liked = models.BooleanField(default=False, blank=True)
    #If user dislikes a post, turn this field to true
    is_disliked = models.BooleanField(default=False, blank=True)
    #This field contains comments made on user post
    comment = models.TextField(blank=True)
    #This field contains the username of the user who leaves feedbac
    username = models.CharField(max_length=100)
    #Many to one field associating Feedback class to Messages class
    message = models.ForeignKey(Messages, on_delete=models.CASCADE)
```

Python framework converts these classes to SQL style data using Object Relation Mapping(ORM)

Data is saved in SQLite3 database.

Data Serialization

Model serializers are written to convert relational data to JSON objects and vice versa

```python
class FeedbackSerializer(serializers.ModelSerializer):

    class Meta:
        model = Feedback
        fields = ['id', 'is_liked', 'is_disliked', 'comment', 'username', 'message']

"""
MessagesSerializer used to serialize Messages class
"""
class MessagesSerializer(serializers.ModelSerializer):
    #Following SerializerMethodField keeps value of status of a message, i.e., whether a
    #If it is False, the post is expired, if it is True, the post is Live
    live_status = serializers.SerializerMethodField('_check_live_status')
    #Following SerializerMethondField contains the remaining time in seconds before a me
    live_time_remaining = serializers.SerializerMethodField('_check_remaining_time')

    #Serializer method that compares the expiry time with current time to determine live
    def _check_live_status(self, messages_object):
        expiration_time = getattr(messages_object, "expiry_in_seconds")
        if (expiration_time > timezone.now()):
            return True
        else:
            return False
```

Facilitates JSON feeds at API endpoints

Data Viewsets

Functions to facilitate API requests (get, put, update, retrieve, delete)

```python
class SearchExpiredMessageByTopic(viewsets.ModelViewSet):
    serializer_class = MessagesSerializer

    def get_queryset(self):
        messages = Messages.objects.all()
        return messages



    def retrieve(self, request, *args, **kwargs):
        parameters = kwargs

        try:
            message = Messages.objects.filter(topic__topic_name__icontains = parameters['pk'])

            serializer = MessagesSerializer(message, many=True)
            retrieved_messages = serializer.data
            #Using dictionary to create returned object as I am utilising live_status field which is
            #be used in the filter function.
            d= {}
            i = 1
            #Iterating through all messages and adding messages to dictionary that are expired, flag
            for msg in retrieved_messages:

                if not msg["live_status"]:
                    key_name = 'Expired message number: ' + str(i)
                    d[key_name] = msg
```

```python
##Registering messaging viewsets here to router
router.register('message', MessagesViewset, basename = 'Messages')
router.register('feedback', FeedbackViewset, basename = 'Feedback')
router.register('topic', TopicsViewset, basename = 'Topics')
router.register('sortedmessages', MessagesSortedByInteractionViewSet,
router.register('messagebytopic', SearchMessageByTopic, basename = 'M
router.register('messagebytopicsorted', SearchMessageByTopicSorted, b
router.register('expiredmessagebytopic', SearchExpiredMessageByTopic,
```

Functions that process user CRUD requests

Off site backup

Add-on service written in python that fetches all records from main API backend and posts it to cloud based MySQL database

We have also written a user application in python that interacts with our API to demonstrate functionalities

END OF PRESENTATION

Any Questions?