



Fernando Barreto
Nelson Netto
Victor Hugo

Modo de Desenvolvimento

- ▶ Uso de diagramas UML para planejar a engenharia do sistema,
- ▶ Desenvolvimento de interfaces gráficas com HTML, CSS e JavaScript,
- ▶ Aplicação do paradigma orientado a objetos,
- ▶ Utilização das models do Django para sincronizar dados com o banco SQLite,
- ▶ Organização do sistema baseada em estrutura de dados e ordenação dinâmica,
- ▶ Avaliação do sistema com base nos princípios de análise de complexidade.



Classes

- ▶ O sistema se dividiu, basicamente entre duas classes principais; Grupo e Usuário

```
class User(models.Model):
    picture = models.ImageField(upload_to=user_profile_image_path, blank=True, null=True, default='media/anonymous.png')
    name = models.TextField(max_length=80, default = 'name')
    id = models.TextField(max_length=20, default = 'id', primary_key=True)
    description = models.TextField(max_length=800, default = 'description')
    password = models.TextField(max_length=20, default = 'password')
    theme = models.TextField(max_length=5, default = 'light')
    language = models.TextField(max_length=2, default = 'pt')

    groupsList = models.ForeignKey('groupLinkedList', on_delete=models.CASCADE, null=True, blank=True)

class Group(models.Model):
    picture = models.ImageField(upload_to=group_image_path, blank=True, null=True, default='media/defBanner.png')
    name = models.TextField(max_length=80, default = 'name')
    id = models.TextField(max_length=20, default = 'id', primary_key=True)
    description = models.TextField(max_length=800, default = 'description')

    adminList = models.ForeignKey('userBinarySearchTree', on_delete=models.CASCADE, null=True, blank=True, related_name='admins')
    membersList = models.ForeignKey('userBinarySearchTree', on_delete=models.CASCADE, null=True, blank=True, related_name='members')
```

- ▶ Além dos atributos principais como nome e foto, as classes mostradas também possuem associações com as classes de árvores e as listas ligadas. O que nos permitiu a organizar, em árvores binárias, a lista de membros e administradores de um grupo em ordem alfabética, e em listas ligadas os grupos cada usuário em formato de uma pilha LIFO para organizar os grupos por ordem de entrada. Essas estruturas são fundamentais para efetuar autenticações ao longo do código.

Listas e Árvores

- ▶ As estruturas utilizadas, por sua vez, são formadas por um nó que se liga a diversos outros, e cada um desses nós representa um usuário único, com todos os dados necessários para o registro de sua jornada de trabalho.

```
class groupListNode(models.Model):
    data = models.ForeignKey('Group', on_delete=models.CASCADE, null=True, blank=True)
    next = models.ForeignKey('groupListNode', on_delete=models.CASCADE, null=True, blank=True)

class groupLinkedList(models.Model):
    head = models.ForeignKey('groupListNode', on_delete=models.CASCADE, null=True, blank=True)
```

```
class userTreeNode(models.Model):
    data = models.ForeignKey('User', on_delete=models.CASCADE, null=True, blank=True, related_name='dataTreeNode')
    state = models.TextField(max_length=1, default = '0')
    schedule = models.ForeignKey('scheduleList', on_delete=models.CASCADE, null=True, blank=True)
    left = models.ForeignKey('userTreeNode', on_delete=models.CASCADE, null=True, blank=True, related_name='leftData')
    right = models.ForeignKey('userTreeNode', on_delete=models.CASCADE, null=True, blank=True, related_name='rightData')

class userBinarySearchTree(models.Model):
    root = models.ForeignKey('userTreeNode', on_delete=models.CASCADE, null=True, blank=True, related_name='userTree')
```

- ▶ A primeira classe é usada unicamente para armazenar um grupo e é uma forma de confirmar para o sistema que o usuário está nele. A segunda classe é a forma que encontramos de, em cada grupo, reservar um espaço único para armazenar os dados de cada usuário. Em **data** coloca-se as informações de seu perfil, em **state** informa-se se ele está ativo ou não e em **schedule** armazenam-se todos os horários em que o usuário pressionou o botão de registro

Gerenciamento de horários

- O gerenciamento de horários é efetuado por uma terceira classe, a **schedule**. Uma lista duplamente ligada que receberá os horários em vários de seus nós e armazenará cada um deles no banco de dados, em um local especialmente reservado para cada usuário.

```
class scheduleListNode(models.Model):
    previous = models.ForeignKey('scheduleListNode', on_delete=models.CASCADE, null=True, blank=True, related_name='previous_node')
    data = models.TextField(max_length=50, null=True, blank=True)
    next = models.ForeignKey('scheduleListNode', on_delete=models.CASCADE, null=True, blank=True, related_name='next_node')

class scheduleList(models.Model):
    head = models.ForeignKey('scheduleListNode', on_delete=models.CASCADE, null=True, blank=True)
```



Métodos e Funções

- Por fim, gostaríamos de apresentar alguns métodos e funções do código, apenas os mais relevantes para a matéria de estrutura de dados, pois o tempo é curto e são muitos os métodos independentes de uma estrutura mais complexa, mas que ainda assim trabalham pelo funcionamento do sistema.

```
def isEmpty(self):  
    return self.head.data is None
```

Verifica se uma estrutura está vazia.

```
def __traverse(self, node):  
    if node is None:  
        return  
    for nextData in self.__traverse(node.next):  
        yield(nextData)  
    yield(node.data)  
  
def __traverse(self, node):  
    if node is None:  
        return  
  
    for leftData in self.__traverse(node.left):  
        yield(leftData)  
  
    yield(node.data)  
  
    for rightData in self.__traverse(node.right):  
        yield(rightData)
```

Usados para percorrer os nós de uma lista ou árvore.

```
def append(self, new_data):  
    if self.isEmpty():  
        self.head.data = new_data  
        self.head.save()  
    else:  
        parent = self  
        current = self.head  
        while current:  
            parent = current  
            current = current.next  
        new_node = scheduleListNode(previous=parent, data=new_data, next=None)  
        new_node.save()  
        parent.next = new_node  
        parent.save()  
    self.save()
```

Adiciona um elemento a uma lista.

```
def clear(self):  
    self.head.data = None  
    self.head.next = None  
    self.head.save()
```

Limpa todos os elementos da lista.

```
def __delete(self, parent, node):
    deleted = node.data

    if node.left:
        if node.right:
            self.__promote_sucessor(node)
        else:
            if parent is self:
                self.root = node.left

            elif parent.left is node:
                parent.left = node.left

            else:
                parent.right = node.left

    else:
        if parent is self:
            if node.right:
                self.root = node.right
            else:
                self.root.data = None
                self.root.save()

        elif parent.left is node:
            parent.left = node.right

        else:
            parent.right = node.right

def __delete(self, goal):
    parent, node = self.search(goal)
    if parent is self:
        if node.next is None:
            self.head.data = None
            self.head.save()
        else:
            self.head = node.next
            self.save()
    else:
        parent.next = node.next
        parent.save()
```

Usado pra deletar o nó de uma árvore ou lista.

```
def __find(self, goal):
    current = self.head
    parent = self
    while (current and goal != current.data.id):
        parent = current
        current = current.next
    return parent, current

def __find(self, goal):
    current = self.root
    parent = self

    while (current and goal != current.data):
        parent = current
        current = (current.left if goal.name < current.data.name else current.right)
    return parent, current
```

Usado para encontrar um nó na lista ou árvore

```
def nodes(self):
    count = 0
    for index in self.traverse():
        count += 1
    return count
```

Usado para contar a quantidade de nós na árvore.

```
def insert(self, data):
    if self.isEmpty():
        self.root.data = data
        thescheduleNode = scheduleListNode(previous=None, data=None, next=None)
        thescheduleNode.save()
        theschedule = scheduleList(head=thescheduleNode)
        theschedule.save()
        self.root.schedule = theschedule
        self.root.state = '0'
        self.root.left = None
        self.root.right = None
        self.root.save()
    else:
        parent, node = self.__find(data)
        if node:
            node.data = data
            return False
        elif data.name < parent.data.name:
            thescheduleNode = scheduleListNode(previous=None, data=None, next=None)
            thescheduleNode.save()
            theschedule = scheduleList(head=thescheduleNode)
            theschedule.save()
            parent.left = userTreeNode(data=data, left = None, right = None, state = '0', schedule = theschedule)
            parent.left.save()
        else:
            thescheduleNode = scheduleListNode(previous=None, data=None, next=None)
            thescheduleNode.save()
            theschedule = scheduleList(head=thescheduleNode)
            theschedule.save()
            parent.right = userTreeNode(data=data, left = None, right = None, state = '0', schedule = theschedule)
            parent.right.save()
        parent.save()
    self.save()
```

Usado para inserir um dado na árvore em ordem alfabética

Fora esses, foram criados métodos para:

- ▶ Renderização de telas
- ▶ Cadastro de usuário
- ▶ Login e autenticação
- ▶ Edição de dados de um usuário
- ▶ Exclusão de usuário
- ▶ Busca por outros usuários
- ▶ Configurar características de interface
- ▶ Criar grupos
- ▶ Buscar grupos
- ▶ Entrar em grupos
- ▶ Alterar dados do grupo (ADM)
- ▶ Apagar grupo (ADM)
- ▶ Promover e Rebaixar usuários (ADM)
- ▶ Registrar horários de entrada e saída
- ▶ Gerar planilhas com os horários da jornada de trabalho



Muito obrigado pela atenção!



Agora vamos para uma demonstração do sistema!