```python
from typing import Dict, Any
from service_desk_manager import ServiceDeskManager
from form_parser import ServiceDeskFormParser
from form_manager import ServiceDeskFormManager
from response import CreateRequestResponseParser

FORM_DIDNT_FETCH_ERROR = "Form has not been fetched and parsed. Please run
`fetch_and_parse_form` first."

class ServiceDeskFormClient:
    """
    A client class for interacting with Atlassian Service Desk forms.

    This class provides utilities to fetch forms, parse them, manage form fields, and
create service desk requests.

    Attributes
    ----------
    base_url : str
        The base URL of the Atlassian Service Desk.
    username : str
        The username used for authentication.
    auth_token : str
        The authentication token.
    service_desk_manager : ServiceDeskManager
        An instance of ServiceDeskManager for managing service desk requests.
    form_manager : ServiceDeskFormManager
        An instance of ServiceDeskFormManager for managing form fields and values.

    Methods
    -------
    fetch_and_parse_form(portal_id: int, request_type_id: int) -> None
        Fetches and parses the form for the given portal and request type ID.
    list_fields() -> None
        Lists all fields in the form.
    list_field_values(field_name: str) -> None
        Lists possible values for a specific field in the form.
    set_form_values(values: Dict[str, Any]) -> Dict[str, Any]
        Sets the values for the form fields.
    create_request(filled_values: Dict[str, Any]) -> CreateRequestResponseParser
        Creates a service desk request with the filled form values.
    """

    def __init__(self, base_url: str, username: str, auth_token: str) -> None:
        """
        Initializes the ServiceDeskFormClient with the provided credentials.
        Parameters
        ----------
        base_url : str
            The base URL of the Atlassian Service Desk.
        username : str
            The username used for authentication.
        auth_token : str
            The authentication token.
        """
        self.base_url = base_url
        self.username = username
        self.auth_token = auth_token
        self.service_desk_manager = ServiceDeskManager(base_url=self.base_url,
username=self.username, auth_token=self.auth_token)
        self.form_manager = None

    def fetch_and_parse_form(self, portal_id: int, request_type_id: int) -> None:
        """
        Fetches and parses the form for the given portal and request type ID.

        Parameters
        ----------
        portal_id : int
            The ID of the service desk portal.
        request_type_id : int
            The ID of the request type.
        """
        form = self.service_desk_manager.fetch_form(portal_id=portal_id,
request_type_id=request_type_id)
        form_obj = ServiceDeskFormParser.parse(form)
        self.form_manager = ServiceDeskFormManager(form_obj)

    def list_fields(self) -> None:
        """
        Lists all fields in the fetched form.

        Raises
        ------
        ValueError
            If the form has not been fetched and parsed.
        """
        if self.form_manager is None:
            raise ValueError(FORM_DIDNT_FETCH_ERROR)
        self.form_manager.list_fields()
    def list_field_values(self, field_name: str) -> None:
        """
        Lists possible values for a specific field in the form.

        Parameters
        ----------
        field_name : str
            The name of the field to list values for.

        Raises
        ------
        ValueError
            If the form has not been fetched and parsed.
        """
        if self.form_manager is None:
            raise ValueError(FORM_DIDNT_FETCH_ERROR)
        self.form_manager.list_field_values(field_name)

    def set_form_values(self, values: Dict[str, Any]) -> Dict[str, Any]:
        """
        Sets the values for the form fields.

        Parameters
        ----------
        values : Dict[str, Any]
            A dictionary where the keys are field names and the values are the values to set
for the fields.

        Returns
        -------
        Dict[str, Any]
            The filled form ready to be submitted as a request.

        Raises
        ------
        ValueError
            If the form has not been fetched and parsed.
        """
        if self.form_manager is None:
            raise ValueError(FORM_DIDNT_FETCH_ERROR)
        return self.form_manager.set_field_values(values)

    def create_request(self, filled_values: Dict[str, Any]) ->
CreateRequestResponseParser:
        """
        Creates a service desk request with the filled form values.
        Parameters
        ----------
        filled_values : Dict[str, Any]
            The filled form values.

        Returns
        -------
        CreateRequestResponseParser
            The response of the create request parsed into a
CreateRequestResponseParser object.
        """
        response_dict = self.service_desk_manager.create_request(filled_values)
        return CreateRequestResponseParser.parse(response_dict)
# %%
NFS_ATLASSIAN_ACCOUNT_URL = "https://naturapay.atlassian.net"
USER_NAME = "victor.souza.tw@naturapay.net"
AUTH_TOKEN = "ATATT3xFfGF09D8JZSCnM5DObhfzSduqNDBUlweizJIaHDVsDQ-
j32q57EgbHyNbSO-yXYheUs14Q5qGhd33_Y2TBSiovqo15m-f2mINCoGpF5biHlgIxCL8
FXqN4QOiRbiuFtLAg7eukYNY3jIYZM8HDYwiksYd_OpsS_IxUhJH_g9Ev3qtK4A=8BC4
1410"

# %%
from service_desk_manager import ServiceDeskManager
service_desk_client =
ServiceDeskManager(base_url=NFS_ATLASSIAN_ACCOUNT_URL,
username=USER_NAME, auth_token=AUTH_TOKEN)

# %%
request_types = service_desk_client.get_request_types(service_desk_id=22,
group_id=114)

# %%
request_types

# %%
form = service_desk_client.fetch_form(portal_id=22, request_type_id=399)

# %%
from form_parser import ServiceDeskFormParser

form_obj = ServiceDeskFormParser.parse(form)

# %%
form_obj.fields
```

```
# %%
from form_manager import ServiceDeskFormManager
form_manager = ServiceDeskFormManager(form_obj)

# %%
form_manager.list_fields()

# %%
form_manager.list_field_values('customfield_10235')

# %%
for env in ["DEV", "HML", "PRD"]:
    summary = f"Solicitação de acesso ao ambiente Databricks {env}"
    tipo_solicitacao = "Adicionar"


# %%
# Example of setting compound field values using labels instead of IDs
tuple_tribe = ('Data', 'Domínio')
filled_values = {
    "summary": "Correção em log do job da tabela cb_status_atual_credito_pay",
    "Tribos/CoE X Squads": tuple_tribe,
    'Status GMUD': "Aguardar Aprovação",
    "Card Number": "TCEC-4137",
    "Participantes | Responsáveis": "Victor Mariano Leite Prado de Souza",
    "Escopo Edição": 'Aplicações',
    "Tipo de Versão": "Patch",
    "SREs Envolvidos": 'Marco Gabriel | marco.gabriel.tw@naturapay.net',
    'Plano de Testes': "Validar se os logs no Kibana estão sendo gerados corretamente.",
    'Data Encerrameto GMUD': "2024-08-21T18:00",
    'Versão do Deploy': "v5.1.11",
    "Componentes | Produtos Envolvidos": "financial-services-refin-cb-status-atual-
credito-pay-job",
    "Benefício para Negócio": "O job da tabela cb_status_atual_credito_pay não está
gerando logs corretamente no Kibana. O objetivo é corrigir o job para que os logs
sejam gerados, para que possamos acompanhar a execução do job adequadamente.",
    "Nome do Repositório (Não colocar url)": "financial-services-refin-cb-status-atual-
credito-pay-job",
    "Riscos de Implantação": "Nenhum.",
    "Plano de Rollback": "Git revert",
    "Riscos de Não implantação": "Não conseguiremos acompanhar a execução do job
da tabela cb_status_atual_credito_pay.",
    'Pós Implantação': "Validar se os logs no Kibana estão sendo gerados corretamente
e fazer dashboard com os dados do job.",
}
```

```
form_filled = form_manager.set_field_values(filled_values)


# %%
import urllib.parse
import json

encoded_url = form_filled.to_request_payload()
parsed_data = urllib.parse.parse_qs(encoded_url)

# Decode any JSON strings within the parsed data
for key, value in parsed_data.items():
    if len(value) == 1 and value[0].startswith('{') and value[0].endswith('}'):
        parsed_data[key] = json.loads(value[0])
    else:
        parsed_data[key] = value[0] if len(value) == 1 else value

# Convert to a JSON string with indentation
json_formatted_str = json.dumps(parsed_data, indent=4, ensure_ascii=False)

# Print the formatted JSON string
print(json_formatted_str)


# %%
# response of service_desk_client.create_request(form_filled)
response_dict = {'reporter': {'email': 'victor.souza.tw@naturapay.net',
  'displayName': 'Victor Mariano Leite Prado de Souza - Thoughtworks',
  'avatarUrl': 'https://avatar-management--avatars.us-west-2.prod.public.atl-
paas.net/712020:376db4a6-0fc6-4237-a217-6ada76d299f3/0d2bbb2e-2270-4a46-8673-
d1090aa4cb18/24',
  'accountId': '712020:376db4a6-0fc6-4237-a217-6ada76d299f3'},
 'participants': [],
 'organisations': [],
 'requestTypeName': 'GMUD - Edição',
 'key': 'GMUD-12396',
 'issueType': '10203',
 'issueTypeName': 'Gestão Mudança',
 'issue': {'id': 116459,
  'key': 'GMUD-12396',
  'reporter': {'email': 'victor.souza.tw@naturapay.net',
   'displayName': 'Victor Mariano Leite Prado de Souza - Thoughtworks',
   'avatarUrl': 'https://avatar-management--avatars.us-west-2.prod.public.atl-
paas.net/712020:376db4a6-0fc6-4237-a217-6ada76d299f3/0d2bbb2e-2270-4a46-8673-
d1090aa4cb18/24',
   'accountId': '712020:376db4a6-0fc6-4237-a217-6ada76d299f3'},
  'participants': [],
  'organisations': [],
  'sequence': 12396,
  'serviceDeskKey': 'gmud',
  'requestTypeName': 'GMUD - Edição',
  'requestTypeId': 435,
  'summary': 'Correção em log do job da tabela cb_status_atual_credito_pay',
  'isNew': True,
  'status': 'Em Plano Execução',
  'date': '2024-08-21T00:21:56-0300',
  'friendlyDate': 'Hoje 12:21 AM',
  'fields': [{'id': 'customfield_10118',
    'label': 'Tribos/CoE X Squads',
    'value': {'text': 'Data - Domínio'}},
   {'id': 'customfield_11003',
    'label': 'Card Number',
    'value': {'adf': '{"type":"doc","version":1,"content":[{"type":"paragraph","content":
[{"type":"text","text":"TCEC-4137"}]}]}'}}],
  'activityStream': [],
  'requestIcon': 10532,
  'iconUrl': 'https://naturapay.atlassian.net/rest/servicedeskapi/requesttype/icon/type/
SD_REQTYPE/id/10532',
  'canBrowse': True,
  'canAttach': True,
  'categoryKey': 'new',
  'creatorAccountId': '712020:376db4a6-0fc6-4237-a217-6ada76d299f3',
  'formKey': ''},
 'canCreateAttachments': True,
 'canCreateIssues': True,
 'canAddComment': True,
 'canViewIssueInJIRA': False,
 'canAddParticipants': False,
 'canRemoveParticipants': True,
 'canSearchParticipants': False,
 'canSignupParticipants': False,
 'canSubmitWithEmailAddress': False,
 'canShareRequest': False,
 'topPanels': [],
 'detailsPanels': [],
 'optionPanels': [],
 'actionSections': [{'params': {'styleClass': 'customer-request-actions'},
   'key': 'custom-customer-request-actions',
   'label': '',
   'items': []}],
 'issueLinkUrl': 'https://naturapay.atlassian.net/browse/GMUD-12396',
 'requestDetailsBaseUrl': '/servicedesk/customer/portal/5/GMUD-12396',
 'customerInvited': False,
 'subscribeAction': 'unsubscribe',
 'approvalStatus': [],
 'workflowTransitions': [],
 'readFileMediaCredentials': {'clientId': '14a26994-a335-446d-84d4-5c6c5d2b746f',
  'endpointUrl': 'https://api.media.atlassian.com',
  'tokenLifespanInSeconds': 3600,
  'tokensWithFiles': []},
 'hasProformaForm': True,
 'isProformaHarmonisationEnabled': False}

# %%
from response import CreateRequestResponseParser

response = CreateRequestResponseParser.parse(response_dict)

# %%
response

# %%




from datetime import datetime
import json
from typing import Any, Dict, List, Optional
from form_parser import ServiceDeskForm, ServiceDeskFormField,
ServiceDeskFormFieldValue


class ServiceDeskFormValidator:
    """
    Validates the field values in a ServiceDeskForm based on their types.

    Methods
    -------
    validate(filled_values: Dict[str, Any], form: ServiceDeskForm) -> None:
        Validates all filled values based on the form's field definitions.
    _validate_dt_field(field: ServiceDeskFormField, value: str) -> None:
        Validates a date-time string for a field.
    _validate_choice_field(field: ServiceDeskFormField, value: Any) -> None:
        Validates a choice field, handling both single and compound fields.
    _validate_text_field(field: ServiceDeskFormField, value: str) -> None:
        Validates a text field.
    _validate_adf_field(field: ServiceDeskFormField, value: str) -> None:
        Validates an Atlassian Document Format (ADF) field.
    _validate_dt(value: str) -> bool:
        Checks if the date-time string is valid.
    _validate_choice(value: str, choices: List[str]) -> bool:
        Checks if the value is within allowed choices.
    _validate_text(value: str, max_length: Optional[int] = None) -> bool:
        Checks if the text is valid, optionally validating length.
    _validate_adf(value: str) -> bool:
        Checks if the ADF string is valid JSON and follows the required structure.
    """

    def validate(self, filled_values: Dict[str, Any], form: ServiceDeskForm) -> None:
        """
        Validates all filled values based on the form's field definitions.

        Parameters
        ----------
        filled_values : Dict[str, Any]
            The dictionary of filled field values to validate.
        form : ServiceDeskForm
            The form object containing field definitions.

        Raises
        ------
        ValueError
            If any field value is invalid according to its type.
        """
        for field_identifier, value in filled_values.items():
            # Check if this is a derived field (e.g., customfield_10118:1)
            field = self._get_field_by_id_or_label(form, field_identifier)

            if not field:
                raise ValueError(f"Field '{field_identifier}' not found in the form.")

            if ':' in field_identifier:  # This is a subfield in a cascading select
                self._validate_cascading_subfield(form, filled_values, field_identifier, value)
            else:
                self._validate_generic_field(field, value)

    def _validate_cascading_subfield(self, form: ServiceDeskForm, filled_values: Dict[str,
Any], field_identifier: str, value: str) -> None:
        """
        Validates a cascadingselect subfield by checking the main field and its associated
subfield value.

        Parameters
        ----------
        form : ServiceDeskForm
            The form containing the fields.
        filled_values : Dict[str, Any]
            The dictionary of filled field values to validate.
        field_identifier : str
            The identifier of the subfield (e.g., 'customfield_10118:1').
        value : str
            The value of the subfield to validate.

        Raises
        ------
        ValueError
            If the main field or subfield value is invalid.
        """
        # Extract the main field ID
        main_field_id = field_identifier.split(':')[0]
        main_field = self._get_field_by_id_or_label(form, main_field_id)

        if not main_field:
            raise ValueError(f"Main field '{main_field_id}' for subfield '{field_identifier}' not
found in the form.")

        # Validate that the main field's value is set correctly
        main_field_value = filled_values.get(main_field_id)
        if not main_field_value:
            raise ValueError(f"Main field '{main_field.label}' with ID '{main_field_id}' is not
set, but subfield '{field_identifier}' is provided.")

        main_value_obj = self._get_value_by_label_or_id(main_field.values,
main_field_value)
        if not main_value_obj:
            raise ValueError(f"Invalid main field value '{main_field_value}' for field
'{main_field.label}' or '{main_field.field_id}'.")

        # Validate the subfield value
        subfield_value_obj = self._get_value_by_label_or_id(main_value_obj.children,
value)
        if not subfield_value_obj:
            available_subfields = [child.label for child in main_value_obj.children]
            raise ValueError(f"Invalid subfield value '{value}' for field '{main_field.label}
(Subfield)' or '{field_identifier}'. "
                             f"Available subfields: {available_subfields}")


    def _validate_generic_field(self, field: ServiceDeskFormField, value: Any) -> None:
        """
        Validates a generic field (non-cascadingselect) based on its type.

        Parameters
        ----------
        field : ServiceDeskFormField
            The field object to validate.
        value : Any
            The value to validate.

        Raises
        ------
        ValueError
            If the value is not valid for the field.
        """
        if field.field_type == 'dt':
            if not self._validate_dt(value):
                raise ValueError(f"Invalid date-time value '{value}' for field '{field.label}' or
'{field.field_id}'.")
        elif field.field_type in ['select', 'radiobuttons', 'multiselect']:
            if not self._validate_choice(value, [v.value for v in field.values]):
                raise ValueError(f"Invalid choice value '{value}' for field '{field.label}' or
'{field.field_id}'.")
        elif field.field_type in ['textarea', 'text']:
            if not self._validate_text(value):
                raise ValueError(f"Invalid text value '{value}' for field '{field.label}' or
'{field.field_id}'.")
        elif field.field_type == 'adf':
            if not self._validate_adf(value):
                raise ValueError(f"Invalid ADF value '{value}' for field '{field.label}' or
'{field.field_id}'.")

    def _validate_dt_field(self, field: ServiceDeskFormField, value: str) -> None:
        """
        Validates a date-time string for a field.

        Parameters
        ----------
        field : ServiceDeskFormField
            The field object.
        value : str
            The value to validate.

        Raises
        ------
        ValueError
            If the value is not a valid date-time string.
        """
        if not self._validate_dt(value):
            raise ValueError(f"Invalid date-time value '{value}' for field '{field.label}' or
'{field.field_id}'.")

    def _validate_choice_field(self, field: ServiceDeskFormField, value: Any) -> None:
        """
        Validates that a value is a valid choice for a field.

        Parameters
        ----------
        field : ServiceDeskFormField
            The field object.
        value : Any
            The value to validate, can be a single value or a tuple for compound fields.

        Raises
        ------
        ValueError
            If the value is not a valid choice.
        """
        if not self._validate_choice(value, [v.value for v in field.values]):
            raise ValueError(f"Invalid choice value '{value}' for field '{field.label}' or
'{field.field_id}'.")


    def _validate_choice(self, value: str, choices: List[str]) -> bool:
        """
        Validates that a value is within a list of allowed choices.

        Parameters
        ----------
        value : str
            The value to validate.
        choices : List[str]
            The list of valid choices.

        Returns
        -------
        bool
            True if the value is within the allowed choices; False otherwise.
        """
        return value in choices

    def _validate_text_field(self, field: ServiceDeskFormField, value: str) -> None:
        """
        Validates a text field.

        Parameters
        ----------
        field : ServiceDeskFormField
            The field object.
        value : str
            The value to validate.

        Raises
        ------
        ValueError
            If the value is not valid text.
        """
        if not self._validate_text(value):
            raise ValueError(f"Invalid text value '{value}' for field '{field.label}' or
'{field.field_id}'.")

    def _validate_adf_field(self, field: ServiceDeskFormField, value: str) -> None:
        """
        Validates an Atlassian Document Format (ADF) field.

        Parameters
        ----------
        field : ServiceDeskFormField
            The field object.
        value : str
            The value to validate.

        Raises
        ------
        ValueError
            If the value is not a valid ADF string.
        """
        if not self._validate_adf(value):
            raise ValueError(f"Invalid ADF value '{value}' for field '{field.label}' or
'{field.field_id}'.")

    def _validate_dt(self, value: str) -> bool:
        """
        Checks if the date-time string is valid.

        Parameters
        ----------
        value : str
            The date-time string to validate.
        Returns
        -------
        bool
            True if the value is a valid date-time string; False otherwise.
        """
        try:
            datetime.strptime(value, "%Y-%m-%dT%H:%M")
            return True
        except ValueError:
            return False

    def _validate_choice(self, value: str, choices: List[str]) -> bool:
        """
        Checks if the value is within allowed choices.

        Parameters
        ----------
        value : str
            The value to validate.
        choices : List[str]
            The list of valid choices.

        Returns
        -------
        bool
            True if the value is within the allowed choices; False otherwise.
        """
        return value in choices

    def _validate_text(self, value: str, max_length: Optional[int] = None) -> bool:
        """
        Checks if the text is valid, optionally validating length.

        Parameters
        ----------
        value : str
            The text value to validate.
        max_length : Optional[int]
            The maximum allowed length for the text.

        Returns
        -------
        bool
            True if the text is valid; False otherwise.
        """
        if max_length is not None and len(value) > max_length:
            return False
        return isinstance(value, str)

    def _validate_adf(self, value: str) -> bool:
        """
        Checks if the ADF string is valid JSON and follows the required structure.

        Parameters
        ----------
        value : str
            The ADF JSON string to validate.

        Returns
        -------
        bool
            True if the value is a valid ADF string; False otherwise.
        """
        try:
            adf = json.loads(value)
            if adf.get("type") == "doc" and isinstance(adf.get("content"), list):
                return True
            return False
        except json.JSONDecodeError:
            return False

    def _get_field_by_id_or_label(self, form: ServiceDeskForm, identifier: str) ->
Optional[ServiceDeskFormField]:
        """
        Retrieves a field by its ID or label from the form.

        Parameters
        ----------
        form : ServiceDeskForm
            The form containing the fields.
        identifier : str
            The ID or label of the field to retrieve.

        Returns
        -------
        Optional[ServiceDeskFormField]
            The ServiceDeskFormField instance if found, None otherwise.
        """
        return next((field for field in form.fields if field.field_id == identifier or field.label ==
identifier), None)
```

```
    def _get_value_by_label_or_id(self, values: List[ServiceDeskFormFieldValue],
identifier: str) -> Optional[ServiceDeskFormFieldValue]:
        """
        Retrieves a value by its label or ID from a list of ServiceDeskFormFieldValue
instances.

        Parameters
        ----------
        values : List[ServiceDeskFormFieldValue]
            The list of ServiceDeskFormFieldValue instances to search.
        identifier : str
            The label or ID of the value to retrieve.

        Returns
        -------
        Optional[ServiceDeskFormFieldValue]
            The ServiceDeskFormFieldValue instance if found, None otherwise.
        """
        return next((value for value in values if value.label == identifier or value.value ==
identifier), None)from dataclasses import dataclass, field
from typing import Dict, Any, List, Optional, Union, Tuple
from field_validator import ServiceDeskFormValidator
from form_parser import ServiceDeskForm, ServiceDeskFormField,
ServiceDeskFormFieldValue
import json
import urllib.parse

@dataclass
class ServiceDeskFormFilled:
    form: ServiceDeskForm
    filled_values: Dict[str, Any] = field(default_factory=dict)

    def to_request_payload(self) -> str:
        """
        Convert the filled values into a URL-encoded string suitable for making the API
request.

        This method handles both regular fields and proformaFormData fields, ensuring
that
        templateId and UUID are included correctly. It also combines nested fields into a
        single field with comma-separated values.

        Returns
        -------
        str
            The URL-encoded payload for the API request.
        """
        # Process all fields and combine results
        regular_fields = self._process_regular_fields()
        proforma_data = self._construct_proforma_data()

        # Add the proformaFormData as a JSON string
        regular_fields["proformaFormData"] = json.dumps(proforma_data)
        regular_fields["projectId"] = str(self.form.project_id)


        # URL-encode the entire payload
        url_encoded_payload = urllib.parse.urlencode(regular_fields)

        return url_encoded_payload

    def _process_regular_fields(self) -> Dict[str, Any]:
        """
        Process regular fields from the filled values.

        Returns
        -------
        Dict[str, Any]
            A dictionary of regular fields with their values.
        """
        regular_fields = {}
        for field in self.form.fields:
            field_id = field.field_id
            if field_id in self.filled_values and not field.is_proforma_field:
                regular_fields[field_id] = self.filled_values[field_id]
        return regular_fields

    def _construct_proforma_data(self) -> Dict[str, Any]:
        """
        Construct the proformaFormData section of the payload.

        Returns
        -------
        Dict[str, Any]
            The proformaFormData section.
        """
        proforma_answers = {}

        for field in self.form.fields:
            field_id = field.field_id
            if field_id in self.filled_values and field.is_proforma_field:
                    proforma_answers[field.proforma_question_id] =
    self._process_proforma_field(field_id, field.field_type)

            return {
                "templateFormId": self.form.template_id,
                "answers": proforma_answers
            }

        def _process_proforma_field(self, field_id: str, field_type: str) -> Dict[str, Any]:
            """
            Process a proforma field and convert it into the appropriate format.

            Parameters
            ----------
            field_id : str
                The ID of the field being processed.
            field_type : str
                The type of the field.

            Returns
            -------
            Dict[str, Any]
                The processed proforma field in the correct format.
            """
            # Logic made for Proforma Form fields that have options
            form_values = self.form.get_field_by_id(field_id).values
            if isinstance(form_values, list) and len(form_values) > 1:
                field_type = "cl"

            value = self.filled_values[field_id]

            # Handle rich text fields with ADF formatting
            if field_type in ["rt", "cd"]:
                return {"adf": self._create_adf_document(value)}

            # Handle choice fields
            elif field_type == "cl":
                # Convert value to the corresponding ID for the choice
                choice_id = self._get_choice_id(field_id, value)
                return {"text": "", "choices": [choice_id]}

            # Handle date-time fields
            elif field_type == "dt":
                date, time = value.split("T")
                return {"date": date, "time": time}
            # Handle simple text fields
            else:
                return {"text": value}

        def _get_choice_id(self, field_id: str, value: str) -> str:
            """
            Retrieve the ID corresponding to a choice value.

            Parameters
            ----------
            field_id : str
                The ID of the field.
            value : str
                The choice value to be converted to its ID.

            Returns
            -------
            str
                The ID corresponding to the choice value.
            """
            field = self.form.get_field_by_id(field_id)
            if not field:
                raise ValueError(f"Field ID '{field_id}' not found.")

            # Search through the field's values to find the matching ID
            for choice in field.values:
                if choice.label == value or choice.value == value:
                    return choice.value

            raise ValueError(f"Value '{value}' not found in choices for field '{field_id}'.")

        def _create_adf_document(self, text: str) -> Dict[str, Any]:
            """
            Create an ADF (Atlassian Document Format) document.

            Parameters
            ----------
            text : str
                The text to include in the ADF document.

            Returns
            -------
            Dict[str, Any]
                A JSON object representing the ADF document.
            """
            return {
                "version": 1,
                "type": "doc",
                "content": [
                    {
                        "type": "paragraph",
                        "content": [
                            {"type": "text", "text": text}
                        ]
                    }
                ]
            }



    class ServiceDeskFormManager:
        """
        Manages the ServiceDeskForm, providing functionality to list fields, list field values,
        validate the form, and set field values to create a ServiceDeskFormFilled instance.

        Attributes
        ----------
        form : ServiceDeskForm
            The ServiceDeskForm instance to manage.

        Methods
        -------
        list_fields() -> List[str]:
            Lists all field labels in the form.

        list_field_values(field_identifier: str, parent_value: Optional[str] = None) -> List[str]:
            Lists all possible values for a given field, identified by either label or ID.
            If the field has nested values, the user can pass a parent value to list the children
    of that value.

        validate(filled_values: Dict[str, Any]) -> bool:
            Validates the filled values according to the required fields in the form.

        set_field_values(filled_values: Dict[str, Any]) -> ServiceDeskFormFilled:
            Sets the provided values for the form fields, including compound fields with
    children,
            and returns a ServiceDeskFormFilled instance.
        """

        def __init__(self, form: ServiceDeskForm):
            """
            Initializes the ServiceDeskFormManager with a ServiceDeskForm.
            Parameters
            ----------
            form : ServiceDeskForm
                The ServiceDeskForm instance to manage.
            """
            self.form = form
            self.validator = ServiceDeskFormValidator()

        def create_request_payload(self, filled_values: Dict[str, Any]) -> Dict[str, Any]:
            """
            Converts the filled values into the body format required by the API request,
            handling both regular fields and proformaFormData fields.

            Parameters
            ----------
            filled_values : Dict[str, Any]
                The dictionary of filled field values.

            Returns
            -------
            Dict[str, Any]
                The payload formatted for the API request.
            """
            regular_fields = {}
            proforma_answers = {}

            for field in self.form.fields:
                field_id = field.field_id

                if field_id in filled_values:
                    if field.is_proforma_field:
                        # Map the proforma question ID to the appropriate answer
                        proforma_answers[field.proforma_question_id] = filled_values[field_id]
                    else:
                        # Regular field processing
                        regular_fields[field_id] = filled_values[field_id]

            # Construct the proformaFormData section
            proforma_data = {
                "templateFormId": self.form.template_id,
                "answers": proforma_answers
            }

            # Complete payload combining regular fields and proformaFormData
            payload = {
                **regular_fields,
                "proformaFormData": json.dumps(proforma_data)
            }

            return payload

        def list_fields(self) -> List[str]:
            """
            Lists all field labels in the form.

            Returns
            -------
            List[str]
                A list of field labels.
            """
            return [{"label": field.label, "id": field.field_id, "type": field.field_type, "description":
    field.description} for field in self.form.fields]

        def list_field_values(self, field_identifier: str, parent_value: Optional[str] = None) ->
    List[str]:
            """
            Lists all possible values for a given field, identified by either label or ID.
            If the field has nested values, the user can pass a parent value to list the children
    of that value.

            Parameters
            ----------
            field_identifier : str
                The label or ID of the field whose values to list.
            parent_value : Optional[str]
                The value of the parent to list its children, if applicable.

            Returns
            -------
            List[str]
                A list of value labels for the given field.
            """
            field = self._get_field_by_id_or_label(field_identifier)

            if not field:
                raise ValueError(f"Field with identifier '{field_identifier}' not found.")

            if parent_value is None:
                return [{"label": value.label, "value": value.value} for value in field.values]
            else:
                parent = self._get_value_by_label(field.values, parent_value)
                if parent:
                    return [{"label": child.label, "value": child.value} for child in parent.children]
                else:
                    raise ValueError(f"Parent value '{parent_value}' not found in field
    '{field_identifier}'.")

        def _get_field_by_id_or_label(self, identifier: str) -> Optional[ServiceDeskFormField]:
            """
            Retrieves a field by its ID or label.

            Parameters
            ----------
            identifier : str
                The ID or label of the field to retrieve.

            Returns
            -------
            Optional[ServiceDeskFormField]
                The ServiceDeskFormField instance if found, None otherwise.
            """
            return next((field for field in self.form.fields if field.field_id == identifier or field.label
    == identifier), None)

        def _get_value_by_label_or_id(self, values: List[ServiceDeskFormFieldValue],
    identifier: str) -> Optional[ServiceDeskFormFieldValue]:
            """
            Retrieves a value by its label or ID from a list of ServiceDeskFormFieldValue
    instances.

            Parameters
            ----------
            values : List[ServiceDeskFormFieldValue]
                The list of ServiceDeskFormFieldValue instances to search.
            identifier : str
                The label or ID of the value to retrieve.

            Returns
            -------
            Optional[ServiceDeskFormFieldValue]
                The ServiceDeskFormFieldValue instance if found, None otherwise.
            """
            return next((value for value in values if value.label == identifier or value.value ==
    identifier), None)

        def validate(self, filled_values: Dict[str, Any]) -> bool:
            """
            Validates the filled values according to the required fields in the form.

            Parameters
            ----------
            filled_values : Dict[str, Any]
                The dictionary of filled field values to validate.

            Returns
            -------
            bool
                True if the filled values are valid, otherwise raises an exception.
            """
            required_fields = [field.field_id for field in self.form.fields if field.is_required()]
            missing_fields = set(required_fields) - set(filled_values.keys())

            if missing_fields:
                raise ValueError(f"Missing required fields: {missing_fields}")

            # Validate the consistency of the filled values
            self.validator.validate(filled_values, self.form)
            return True

        def set_field_values(self, filled_values: Dict[str, Any]) -> ServiceDeskFormFilled:
            """
            Sets the provided values for the form fields, including compound fields with
    children,
            and returns a ServiceDeskFormFilled instance.

            Parameters
            ----------
            filled_values : Dict[str, Any]
                The dictionary of filled field values, identified by either labels or IDs.

            Returns
            -------
            ServiceDeskFormFilled
                An instance of ServiceDeskFormFilled with the provided values.
            """
            # Convert labels to IDs if necessary
            filled_values = self._convert_labels_to_ids(filled_values)

            # Validate the filled values
            self.validate(filled_values)

            # Flatten compound fields with children
            flat_filled_values = self._flatten_field_values(filled_values)
            # Create the ServiceDeskFormFilled instance
            form_filled = ServiceDeskFormFilled(form=self.form,
    filled_values=flat_filled_values)

            return form_filled

        def _get_value_by_label(self, values: List[ServiceDeskFormFieldValue], label: str) ->
    Optional[ServiceDeskFormFieldValue]:
            """
            Retrieves a value by its label from a list of ServiceDeskFormFieldValue instances.

            Parameters
            ----------
            values : List[ServiceDeskFormFieldValue]
                The list of ServiceDeskFormFieldValue instances to search.
            label : str
                The label of the value to retrieve.

            Returns
            -------
            Optional[ServiceDeskFormFieldValue]
                The ServiceDeskFormFieldValue instance if found, None otherwise.
            """
            return next((value for value in values if value.label == label), None)

        def _convert_labels_to_ids(self, filled_values: Dict[str, Any]) -> Dict[str, Any]:
            """
            Converts field labels to IDs and value labels to value IDs in the filled values
    dictionary.

            Parameters
            ----------
            filled_values : Dict[str, Any]
                The dictionary of filled field values, which may contain labels instead of IDs.

            Returns
            -------
            Dict[str, Any]
                The dictionary with labels converted to IDs.
            """
            converted_values = {}
            for field_identifier, value in filled_values.items():
                field = self._get_field_by_id_or_label(field_identifier)
                if not field:
                    raise ValueError(f"Field '{field_identifier}' not found in the form.")
                # Ensure correct handling of compound fields and label-to-ID conversion
                if field.field_type in ['cascadingselect', 'select', 'radiobuttons', 'multiselect']:
                    if self._is_compound_field_value(value):
                        converted_values.update(self._convert_compound_field(field, value))
                    else:
                        converted_values[field.field_id] = self._convert_single_field(field, value)
                else:
                    # For text fields and other simple types, directly assign the value
                    converted_values[field.field_id] = value

            return converted_values


        def _convert_single_field(self, field: ServiceDeskFormField, value: Union[str,
    List[str]]) -> Union[str, List[str]]:
            """
            Converts a single field's value(s) from a label(s) to an ID(s).

            Parameters
            ----------
            field : ServiceDeskFormField
                The field to which the value belongs.
            value : Union[str, List[str]]
                The value label(s) to convert.

            Returns
            -------
            Union[str, List[str]]
                The value ID(s) corresponding to the label(s).
            """
            if isinstance(value, list):
                # Handle multi-select fields
                return [self._convert_single_value(field, val) for val in value]
            else:
                # Handle single value
                return self._convert_single_value(field, value)

        def _convert_single_value(self, field: ServiceDeskFormField, value: str) -> str:
            """
            Converts a single value label to its corresponding ID.

            Parameters
            ----------
            field : ServiceDeskFormField
                The field to which the value belongs.
            value : str
                The value label to convert.

            Returns
            -------
            str
                The value ID corresponding to the label.
            """
            # Check if the field has predefined values, otherwise, return the value as is
            if field.values:
                value_obj = self._get_value_by_label_or_id(field.values, value)
                if not value_obj:
                    raise ValueError(f"Invalid value '{value}' for field '{field.label}' or
    '{field.field_id}'.")
                return value_obj.value
            else:
                return value


        def _convert_compound_field(self, field: ServiceDeskFormField, value:
    Union[Tuple[str, str], List[str]]) -> Dict[str, str]:
            """
            Converts a compound field's values from labels to IDs.

            Parameters
            ----------
            field : ServiceDeskFormField
                The field to which the values belong.
            value : Union[Tuple[str, str], List[str]]
                The main and sub value labels to convert.

            Returns
            -------
            Dict[str, str]
                A dictionary with the main value and sub value IDs.
            """
            main_value, sub_value = value
            main_value_obj = self._get_value_by_label_or_id(field.values, main_value)
            sub_value_obj = self._get_value_by_label_or_id(main_value_obj.children,
    sub_value) if main_value_obj else None

            if not main_value_obj or not sub_value_obj:
                raise ValueError(f"Invalid compound value '{main_value}, {sub_value}' for field
    '{field.label}' or '{field.field_id}'.")

            return {
                field.field_id: main_value_obj.value,
                f"{field.field_id}:1": sub_value_obj.value
            }

        def _is_compound_field_value(self, value: Any) -> bool:
            """
            Checks if the provided value is a compound field value.

            Parameters
            ----------
            value : Any
                The value to check.

            Returns
            -------
            bool
                True if the value is a tuple with two elements, indicating a compound field; False
    otherwise.
            """
            return isinstance(value, tuple) and len(value) == 2

        def _flatten_field_values(self, filled_values: Dict[str, Any]) -> Dict[str, Any]:
            """
            Flattens compound field values with children into a format that can be easily
            processed by the Service Desk API.

            Parameters
            ----------
            filled_values : Dict[str, Any]
                The dictionary of filled field values.

            Returns
            -------
            Dict[str, Any]
                The flattened dictionary of filled field values.
            """
            flat_values = {}
            for field_id, value in filled_values.items():
                if isinstance(value, (tuple, list)) and len(value) == 2:
                    main_value, sub_value = value
                    flat_values[field_id] = main_value
                    flat_values[f"{field_id}:1"] = sub_value
                else:
                    flat_values[field_id] = value

            return flat_values
```

```
from typing import Dict, Any, List, Optional
from dataclasses import dataclass, field

@dataclass
class ServiceDeskFormFieldValue:
    """
    Data class representing a value within a Service Desk form field.

    Attributes
    ----------
    value : str
        The value identifier.
    label : str
        The label associated with this value.
    selected : bool
        Whether this value is pre-selected.
    children : List['ServiceDeskFormFieldValue']
        Child values that depend on this value.
    """

    value: str
    label: str
    selected: bool
    children: List['ServiceDeskFormFieldValue'] = field(default_factory=list)
    additional_data: Dict[str, Any] = field(default_factory=dict)

    def add_child(self, child_value: 'ServiceDeskFormFieldValue') -> None:
        """
        Add a child value to this value.

        Parameters
        ----------
        child_value : ServiceDeskFormFieldValue
            The child value to be added.
        """
        self.children.append(child_value)

    def has_children(self) -> bool:
        """
        Check if this value has child values.

        Returns
        -------
        bool
            True if this value has children, False otherwise.
        """
        return len(self.children) > 0

@dataclass
class ServiceDeskFormField:
    """
    Data class representing a field in a Service Desk form.

    Attributes
    ----------
    field_type : str
        The type of the field (e.g., 'text', 'cascadingselect', 'textarea').
    field_id : str
        The unique identifier for the field.
    field_config_id : str
        The configuration ID for the field (may be empty).
    label : str
        The label of the field displayed in the UI.
    description : str
        A brief description of the field's purpose.
    description_html : str
        HTML-formatted description of the field.
    required : bool
        Whether the field is required or optional.
    displayed : bool
        Whether the field is displayed in the form.
    preset_values : List[Any]
        A list of preset values that may be pre-selected or pre-filled.
    values : List[ServiceDeskFormFieldValue]
        A list of possible values for this field, potentially hierarchical.
    renderer_type : Optional[str]
        The renderer type if the field is a textarea.
    auto_complete_url : Optional[str]
        The URL used for autocomplete in case of organisationpicker fields.
    depends_on : Optional[str]
        The ID of another field that this field depends on.
    children : List['ServiceDeskFormField']
        A list of child fields that depend on this field.
    is_proforma_field : bool
        Indicates whether this field is a proforma field.
    proforma_question_id : Optional[str]
        The question ID associated with this field if it's a proforma field.
    """

    field_type: str
    field_id: str
    field_config_id: str
```

```
label: str
description: str
description_html: str
required: bool
displayed: bool
preset_values: List[Any] = field(default_factory=list)
values: List[ServiceDeskFormFieldValue] = field(default_factory=list)
renderer_type: Optional[str] = None
auto_complete_url: Optional[str] = None
depends_on: Optional[str] = None
children: List['ServiceDeskFormField'] = field(default_factory=list)
is_proforma_field: bool = False
proforma_question_id: Optional[str] = None

def is_required(self) -> bool:
    """
    Check if the field is required.

    Returns
    -------
    bool
        True if the field is required, False otherwise.
    """
    return self.required

def has_autocomplete(self) -> bool:
    """
    Check if the field has an autocomplete URL.

    Returns
    -------
    bool
        True if the field has an autocomplete URL, False otherwise.
    """
    return self.auto_complete_url is not None

def add_child(self, child_field: 'ServiceDeskFormField') -> None:
    """
    Add a child field to this field.

    Parameters
    ----------
    child_field : ServiceDeskFormField
        The child field to be added to this field.
    """
    self.children.append(child_field)
    def get_children(self) -> List['ServiceDeskFormField']:
        """
        Get the list of child fields.

        Returns
        -------
        List[ServiceDeskFormField]
            A list of child fields dependent on this field.
        """
        return self.children

    def is_dependent(self) -> bool:
        """
        Check if this field is dependent on another field.

        Returns
        -------
        bool
            True if this field has a dependency, False otherwise.
        """
        return self.depends_on is not None

@dataclass
class ServiceDeskForm:
    """
    Data class representing a Service Desk form.

    Attributes
    ----------
    portal_name : str
        The name of the portal to which the form belongs.
    portal_description : str
        A brief description of the portal's purpose.
    form_name : str
        The name of the form.
    form_description_html : str
        HTML-formatted description of the form.
    fields : List[ServiceDeskFormField]
        A list of fields that make up the form.
    updated_at : Optional[str]
        The last updated timestamp of the form.
    template_id : Optional[int]
        The ID of the template associated with the form.
    template_form_uuid : Optional[str]
        The UUID of the template form.
```

```
"""
id: str
service_desk_id: str
request_type_id: str
project_id: str
portal_name: str
portal_description: str
form_name: str
form_description_html: str
fields: List[ServiceDeskFormField] = field(default_factory=list)
updated_at: Optional[str] = None
template_id: Optional[int] = None
template_form_uuid: Optional[str] = None
atl_token: Optional[str] = None

def add_field(self, field: ServiceDeskFormField) -> None:
    """
    Add a new field to the form.

    Parameters
    ----------
    field : ServiceDeskFormField
        The field to be added to the form.
    """
    self.fields.append(field)

def get_required_fields(self) -> List[ServiceDeskFormField]:
    """
    Get a list of all required fields in the form.

    Returns
    -------
    List[ServiceDeskFormField]
        A list of required fields.
    """
    return [field for field in self.fields if field.is_required()]

def get_field_by_id(self, field_id: str) -> Optional[ServiceDeskFormField]:
    """
    Get a field by its ID.

    Parameters
    ----------
    field_id : str
        The ID of the field to retrieve.
        Returns
        -------
        Optional[ServiceDeskFormField]
            The field with the given ID, or None if not found.
        """
        return next((field for field in self.fields if field.field_id == field_id), None)

    def has_autocomplete_fields(self) -> bool:
        """
        Check if the form contains any fields with autocomplete functionality.

        Returns
        -------
        bool
            True if any field in the form has autocomplete, False otherwise.
        """
        return any(field.has_autocomplete() for field in self.fields)

    def get_dependent_fields(self) -> List[ServiceDeskFormField]:
        """
        Get a list of all fields that have dependencies on other fields.

        Returns
        -------
        List[ServiceDeskFormField]
            A list of fields with dependencies.
        """
        return [field for field in self.fields if field.is_dependent()]


class ServiceDeskFormParser:
    """
    Class to parse JSON responses from the Atlassian Service Desk API
    and convert them into instances of ServiceDeskForm, ServiceDeskFormField,
    and ServiceDeskFormFieldValue.

    Methods
    -------
    parse(json_data: Dict[str, Any]) -> ServiceDeskForm
        Parses the JSON data and returns a ServiceDeskForm object.
    _parse_field(field_data: Dict[str, Any]) -> List[ServiceDeskFormField]
        Parses a single field's data and returns a list of ServiceDeskFormField objects,
        including the main field and subfields.
    _parse_values(values_data: List[Dict[str, Any]], parent_field_id: str = "") ->
List[ServiceDeskFormFieldValue]
        Parses a list of values and returns a list of ServiceDeskFormFieldValue objects.
    _parse_proforma_fields(proforma_data: Dict[str, Any]) -> List[ServiceDeskFormField]
        Parses proforma fields and returns a list of ServiceDeskFormField objects.
    _parse_cascadingselect_field(field: ServiceDeskFormField) ->
List[ServiceDeskFormField]
        Parses and adds subfields for a cascadingselect field.
    """

    @staticmethod
    def parse(json_data: Dict[str, Any]) -> ServiceDeskForm:
        """
        Parses the JSON data and returns a ServiceDeskForm object.

        Parameters
        ----------
        json_data : Dict[str, Any]
            The JSON data from the Atlassian Service Desk API.

        Returns
        -------
        ServiceDeskForm
            An instance of ServiceDeskForm containing parsed data.
        """
        form_id = json_data['portal']['id']
        request_type_id = json_data['reqCreate']['id']
        service_desk_id = json_data['portal']['serviceDeskId']
        project_id = json_data['portal']['projectId']
        portal_name = json_data['portal']['name']
        portal_description = json_data['portal']['description']
        form_name = json_data['reqCreate']['form']['name']
        form_description_html = json_data['reqCreate']['form']['descriptionHtml']

        fields_data = json_data['reqCreate']['fields']
        fields = []
        for field_data in fields_data:
            if 'autoCompleteUrl' not in field_data.keys() or field_data['autoCompleteUrl'] ==
"":
                parsed_fields = ServiceDeskFormParser._parse_field(field_data)
                fields.extend(parsed_fields)

        if 'proformaTemplateForm' in json_data['reqCreate']:
            proforma_fields = ServiceDeskFormParser._parse_proforma_fields(
                json_data['reqCreate']['proformaTemplateForm']
            )
            fields.extend(proforma_fields)

        parsed_fields = ServiceDeskFormParser._parse_autocomplete_fields(json_data)
        fields.extend(parsed_fields)

        proforma_template_form = json_data['reqCreate'].get('proformaTemplateForm', {})
        updated_at = proforma_template_form.get('updated')

        design_settings = proforma_template_form.get('design', {}).get('settings', {})
        template_id = design_settings.get('templateId')
        template_form_uuid = design_settings.get('templateFormUuid')

        return ServiceDeskForm(
            id=form_id,
            project_id=project_id,
            request_type_id=request_type_id,
            service_desk_id=service_desk_id,
            portal_name=portal_name,
            portal_description=portal_description,
            form_name=form_name,
            form_description_html=form_description_html,
            fields=fields,
            updated_at=updated_at,
            template_id=template_id,
            template_form_uuid=template_form_uuid,
            atl_token=json_data.get('xsrfToken', None)
        )

    @staticmethod
    def _parse_field(field_data: Dict[str, Any]) -> List[ServiceDeskFormField]:
        """
        Parses a single field's data and returns a list of ServiceDeskFormField objects,
        including the main field and any subfields if they exist.

        Parameters
        ----------
        field_data : Dict[str, Any]
            The JSON data for a single field.

        Returns
        -------
        List[ServiceDeskFormField]
            A list of ServiceDeskFormField objects, including the main field and subfields.
        """
        field_type = field_data['fieldType']
        field_id = field_data['fieldId']
        field_config_id = field_data.get('fieldConfigId', '')
        label = field_data['label']
        description = field_data.get('description', '')
        description_html = field_data.get('descriptionHtml', '')
        required = field_data['required']
        displayed = field_data['displayed']
        preset_values = field_data.get('presetValues', [])

        values_data = field_data.get('values', [])
        values = ServiceDeskFormParser._parse_values(values_data, field_id)

        renderer_type = field_data.get('rendererType')
        auto_complete_url = field_data.get('autoCompleteUrl')
        depends_on = field_data.get('depends_on')

        main_field = ServiceDeskFormField(
            field_type=field_type,
            field_id=field_id,
            field_config_id=field_config_id,
            label=label,
            description=description,
            description_html=description_html,
            required=required,
            displayed=displayed,
            preset_values=preset_values,
            values=values,
            renderer_type=renderer_type,
            auto_complete_url=auto_complete_url,
            depends_on=depends_on,
            children=[]
        )

        if field_type == 'cascadingselect':
            return ServiceDeskFormParser._parse_cascadingselect_field(main_field)
        else:
            return [main_field]

    @staticmethod
    def _parse_autocomplete_values(values_data: Any) ->
List[ServiceDeskFormFieldValue]:
        """
        Transforms a single result entry from the original JSON format to the new format.

        Parameters
        ----------
        values_data : Dict[str, Any]
            A single result entry from the original JSON.

        Returns
        -------
        Dict[str, Any]
            The transformed values_data entry.
        """
        values = []
        for value_data in values_data["results"]:
            object_type = value_data["objectType"]
            attributes = value_data["attributes"][0]
            value_dict = {
                "value": value_data["objectId"],
                "label": value_data["label"],
                "additional_data": {
                    "workspaceId": value_data["workspaceId"],
                    "objectKey": value_data["objectKey"],
                    "objectType": {
                        "objectTypeId": object_type["objectTypeId"],
                        "id": object_type["id"],
                        "name": object_type["name"],
                        "description": object_type["description"]
                    },
                    "objectTypeAttributeId": attributes["objectTypeAttributeId"],
                    "objectTypeAttributeName": attributes["objectTypeAttribute"]["name"],
                    "objectTypeAttributeType": attributes["objectTypeAttribute"]["type"],
                    "objectTypeAttributeDescription": attributes["objectTypeAttribute"]
["description"],
                    "objectTypeAttributeValues": attributes["objectAttributeValues"]
                },
                "selected": False,
                "children": [],
            }
            field_value = ServiceDeskFormFieldValue(**value_dict)
            values.append(field_value)
        return values

    @staticmethod
    def _parse_autocomplete_fields(values_data: Dict[str, Any]) ->
List[ServiceDeskFormField]:
        """
        Parses Proforma fields and returns a list of ServiceDeskFormField objects.

        Parameters
        ----------
        proforma_data : Dict[str, Any]
            The JSON data containing Proforma fields.

        Returns
        -------
        List[ServiceDeskFormField]
            A list of ServiceDeskFormField objects parsed from Proforma fields.
        """
        autocomplete_fields = []
        fields = values_data['reqCreate']['fields']
        for field in fields:
            if 'autoCompleteUrl' in field.keys() and field['autoCompleteUrl'] != "" and
field['fieldType'] == 'cmdbobjectpicker':
                is_proforma_field = False
                proforma_question_id = None
                autocomplete_values = [field_data for field_data in values_data['reqCreate']
['autocompleteOptions'] if field_data['fieldId'] == field['fieldId']][0]
                values = ServiceDeskFormParser._parse_autocomplete_values(
                    autocomplete_values,
                )
                field_field = ServiceDeskFormField(
                    field_type=field.get("fieldType", ""),
                    field_id=field.get("fieldId", ""),
                    field_config_id='',
                    label=field.get("label"),
                    description=field.get('description', ""),
                    description_html='',
                    required=field.get('required', False),
                    displayed=field.get('displayed', False),
                    preset_values=field.get('presetValues', []),
                    values=[value for value in values if value],
                    is_proforma_field=is_proforma_field,
                    proforma_question_id=proforma_question_id
                )
                autocomplete_fields.append(field_field)
        return autocomplete_fields


    @staticmethod
    def _parse_proforma_values(values_data: List[Dict[str, Any]]) ->
List[ServiceDeskFormFieldValue]:
        """
        Parses a list of values and returns a list of ServiceDeskFormFieldValue objects.

        Parameters
        ----------
        values_data : List[Dict[str, Any]]
            The JSON data for a list of values.
        parent_field_id : str, optional
            The ID of the parent field, used to generate subfield IDs.
        Returns
        -------
        List[ServiceDeskFormFieldValue]
            A list of ServiceDeskFormFieldValue objects.
        """
        values = []
        for value_data in values_data:
            value = value_data['id']
            label = value_data['label']

            field_value = ServiceDeskFormFieldValue(
                value=value,
                label=label,
                selected=None,
                children=None
            )
            values.append(field_value)
        return values

    @staticmethod
    def _parse_values(values_data: List[Dict[str, Any]], parent_field_id: str = "") ->
List[ServiceDeskFormFieldValue]:
        """
        Parses a list of values and returns a list of ServiceDeskFormFieldValue objects.

        Parameters
        ----------
        values_data : List[Dict[str, Any]]
            The JSON data for a list of values.
        parent_field_id : str, optional
            The ID of the parent field, used to generate subfield IDs.

        Returns
        -------
        List[ServiceDeskFormFieldValue]
            A list of ServiceDeskFormFieldValue objects.
        """
        values = []
        for index, value_data in enumerate(values_data):
            value = value_data['value']
            label = value_data['label']
            selected = value_data.get('selected', False)

            children_data = value_data.get('children', [])
            children = ServiceDeskFormParser._parse_values(children_data,
f"{parent_field_id}:{index+1}") if children_data else []

            field_value = ServiceDeskFormFieldValue(
                value=value,
                label=label,
                selected=selected,
                children=children
            )
            values.append(field_value)
        return values

    @staticmethod
    def _parse_proforma_fields(proforma_data: Dict[str, Any]) ->
List[ServiceDeskFormField]:
        """
        Parses Proforma fields and returns a list of ServiceDeskFormField objects.

        Parameters
        ----------
        proforma_data : Dict[str, Any]
            The JSON data containing Proforma fields.

        Returns
        -------
        List[ServiceDeskFormField]
            A list of ServiceDeskFormField objects parsed from Proforma fields.
        """
        fields = []
        questions = proforma_data.get('design', {}).get('questions', {})
        for question_id, question_data in questions.items():
            field_type = question_data['type']
            field_id = question_data.get('jiraField', question_id)
            label = question_data['label']
            description = question_data.get('description', '')
            required = question_data.get('validation', {}).get('rq', False)
            values = proforma_data["proformaFieldOptions"]["fields"].get(field_id, [])

            field = ServiceDeskFormField(
                field_type=field_type,
                field_id=field_id,
                field_config_id='',
                label=label,
                description=description,
                description_html='',
                required=required,
                displayed=True,
                preset_values=[],
                values=ServiceDeskFormParser._parse_proforma_values(values),
                is_proforma_field=True,
                proforma_question_id=question_id
            )
            fields.append(field)
        return fields

    @staticmethod
    def _parse_cascadingselect_field(field: ServiceDeskFormField) ->
List[ServiceDeskFormField]:
        """
        Parses and adds subfields for a cascadingselect field.

        Parameters
        ----------
        field : ServiceDeskFormField
            The main cascadingselect field.

        Returns
        -------
        List[ServiceDeskFormField]
            A list containing the main field and its subfield.
        """
        fields = [field]

        if field.field_type == 'cascadingselect' and field.values:
            subfield_id = f"{field.field_id}:1"
            subfield_label = f"{field.label} (Subfield)"
            subfield = ServiceDeskFormField(
                field_type=field.field_type,
                field_id=subfield_id,
                field_config_id=field.field_config_id,
                label=subfield_label,
                description=field.description,
                description_html=field.description_html,
                required=field.required,
                displayed=field.displayed,
                preset_values=field.preset_values,
                values=[],
                renderer_type=field.renderer_type,
                auto_complete_url=field.auto_complete_url,
                depends_on=field.field_id,
                children=[]
            )
            fields.append(subfield)
        return fields
from dataclasses import dataclass, field
from typing import List, Dict, Optional

@dataclass
class Reporter:
    email: str
    display_name: str
    avatar_url: str
    account_id: str

@dataclass
class IssueField:
    id: str
    label: str
    value: Dict

@dataclass
class Issue:
    id: int
    key: str
    reporter: Reporter
    participants: List[str]
    organisations: List[str]
    sequence: int
    service_desk_key: str
    request_type_name: str
    request_type_id: int
    summary: str
    is_new: bool
    status: str
    date: str
    friendly_date: str
    fields: List[IssueField]
    activity_stream: List[str] = field(default_factory=list)
    request_icon: int = 0
    icon_url: str = ""
    can_browse: bool = True
    can_attach: bool = True
    category_key: str = ""
    creator_account_id: str = ""
    form_key: str = ""

@dataclass
class CreateRequestResponse:
    reporter: Reporter
    request_type_name: str
    key: str
    issue_type: str
    issue_type_name: str
    issue: Issue
    can_create_issues: bool
    can_add_comment: bool
    issue_link_url: str
    request_details_base_url: str

class CreateRequestResponseParser:
    """
    Parser class to convert JSON response into CreateRequestResponse dataclass.

    Methods
    -------
    parse(data: Dict) -> CreateRequestResponse
        Parses the JSON data into a CreateRequestResponse object.
    _parse_reporter(reporter_data: Dict) -> Reporter
        Parses the reporter information.
    _parse_issue_field(field_data: Dict) -> IssueField
        Parses a single issue field.
    _parse_issue(issue_data: Dict) -> Issue
        Parses the issue information.
    """

    @staticmethod
    def parse(data: Dict) -> CreateRequestResponse:
        """
        Parses the JSON data into a CreateRequestResponse object.

        Parameters
        ----------
        data : Dict
            The JSON data to parse.

        Returns
        -------
        CreateRequestResponse
            The parsed CreateRequestResponse object.
        """
        reporter = CreateRequestResponseParser._parse_reporter(data['reporter'])
        issue = CreateRequestResponseParser._parse_issue(data['issue'])

        return CreateRequestResponse(
            reporter=reporter,
            request_type_name=data.get('requestTypeName', ''),
            key=data.get('key', ''),
            issue_type=data.get('issueType', ''),
            issue_type_name=data.get('issueTypeName', ''),
            issue=issue,
            can_create_issues=data.get('canCreateIssues', False),
            can_add_comment=data.get('canAddComment', False),
            issue_link_url=data.get('issueLinkUrl', ''),
            request_details_base_url=data.get('requestDetailsBaseUrl', '')
        )

    @staticmethod
    def _parse_reporter(reporter_data: Dict) -> Reporter:
        """
        Parses the reporter information.

        Parameters
        ----------
        reporter_data : Dict
            The reporter data to parse.

        Returns
        -------
        Reporter
            The parsed Reporter object.
        """
        return Reporter(
            email=reporter_data.get('email', ''),
            display_name=reporter_data.get('displayName', ''),
            avatar_url=reporter_data.get('avatarUrl', ''),
            account_id=reporter_data.get('accountId', '')
        )

    @staticmethod
    def _parse_issue_field(field_data: Dict) -> IssueField:
        """
        Parses a single issue field.

        Parameters
        ----------
        field_data : Dict
            The issue field data to parse.

        Returns
        -------
```

```
        IssueField
            The parsed IssueField object.
        """
        return IssueField(
            id=field_data.get('id', ''),
            label=field_data.get('label', ''),
            value=field_data.get('value', {})
        )

    @staticmethod
    def _parse_issue(issue_data: Dict) -> Issue:
        """
        Parses the issue information.

        Parameters
        ----------
        issue_data : Dict
            The issue data to parse.

        Returns
        -------
        Issue
            The parsed Issue object.
        """
        reporter = CreateRequestResponseParser._parse_reporter(issue_data['reporter'])
        fields = [CreateRequestResponseParser._parse_issue_field(f) for f in
issue_data.get('fields', [])]

        return Issue(
            id=issue_data.get('id', 0),
            key=issue_data.get('key', ''),
            reporter=reporter,
            participants=issue_data.get('participants', []),
            organisations=issue_data.get('organisations', []),
            sequence=issue_data.get('sequence', 0),
            service_desk_key=issue_data.get('serviceDeskKey', ''),
            request_type_name=issue_data.get('requestTypeName', ''),
            request_type_id=issue_data.get('requestTypeId', 0),
            summary=issue_data.get('summary', ''),
            is_new=issue_data.get('isNew', False),
            status=issue_data.get('status', ''),
            date=issue_data.get('date', ''),
            friendly_date=issue_data.get('friendlyDate', ''),
            fields=fields,
            activity_stream=issue_data.get('activityStream', []),
            request_icon=issue_data.get('requestIcon', 0),
            icon_url=issue_data.get('iconUrl', ''),
            can_browse=issue_data.get('canBrowse', True),
            can_attach=issue_data.get('canAttach', True),
            category_key=issue_data.get('categoryKey', ''),
            creator_account_id=issue_data.get('creatorAccountId', ''),
            form_key=issue_data.get('formKey', '')
        )
from atlassian import ServiceDesk, Jira
import base64
import requests
from typing import Any, List, Dict
from form_manager import ServiceDeskFormFilled

# https://naturapay.atlassian.net/servicedesk/customer/portal/6/group/32/create/182 -
Solicitação de Acesso Administrativo
# https://naturapay.atlassian.net/servicedesk/customer/portal/5/group/132/create/437 -
GMUD Inclusão
# https://naturapay.atlassian.net/servicedesk/customer/portal/5/group/132/create/435 -
GMUD Edição
# https://naturapay.atlassian.net/servicedesk/customer/portal/5/group/132/create/436 -
GMUD Rollback

class ServiceDeskRequestError(Exception):
    """
    Custom exception class for errors related to Service Desk requests.

    Attributes
    ----------
    status_code : int
        The HTTP status code returned by the failed request.
    error_message : str
        The error message returned by the failed request.

    Methods
    -------
    __str__():
        Returns a formatted string representation of the error.
    """

    def __init__(self, status_code: int, error_message: str):
        """
        Initializes the ServiceDeskRequestError with the status code and error message.

        Parameters
        ----------
        status_code : int
```

```
            The HTTP status code returned by the failed request.
        error_message : str
            The error message returned by the failed request.
        """
        self.status_code = status_code
        self.error_message = error_message
        super().__init__(self.__str__())

    def __str__(self):
        """
        Returns a formatted string representation of the error.

        Returns
        -------
        str
            A string representation of the error.
        """
        return f"ServiceDeskRequestError: {self.status_code} - {self.error_message}"


def remove_disposable_keys(data, disposable_keys):
    """
    Recursively removes the disposable keys from a dictionary or list.

    Parameters
    ----------
    data : dict or list
        The JSON data (as a dict or list) from which keys should be removed.
    disposable_keys : list
        A list of keys to be removed from the data.

    Returns
    -------
    dict or list
        The cleaned-up data with disposable keys removed.
    """
    if isinstance(data, dict):
        return {k: remove_disposable_keys(v, disposable_keys) for k, v in data.items() if k
not in disposable_keys}
    elif isinstance(data, list):
        return [remove_disposable_keys(item, disposable_keys) for item in data]
    else:
        return data

def clean_response(response):
    """
    Cleans up the JSON response by removing unnecessary keys.

    Parameters
    ----------
    response : dict
        The JSON response to clean.

    Returns
    -------
    dict
        The cleaned response.
    """
    disposable_keys = [
        'key', 'portalBaseUrl', 'onlyPortal',
        'createPermission', 'portalAnnouncement', 'canViewCreateRequestForm',
        'isProjectSimplified', 'mediaApiUploadInformation', 'userLanguageHeader',
        'userLanguageMessageWiki', 'defaultLanguageHeader',
'defaultLanguageMessage',
        'defaultLanguageDisplayName', 'isUsingLanguageSupport', 'translations',
        'callToAction', 'intro', 'instructions', 'icon', 'iconUrl', 'userOrganisations',
        'canBrowseUsers', 'requestCreateBaseUrl', 'requestValidateBaseUrl',
'calendarParams',
        'kbs', 'canRaiseOnBehalf', 'canSignupCustomers', 'canCreateAttachments',
        'attachmentRequiredField', 'hasGroups', 'canSubmitWithEmailAddress',
'showRecaptcha',
        'siteKey', 'hasProformaForm', 'linkedJiraFields', 'portalWebFragments',
'headerPanels',
        'subheaderPanels', 'footerPanels', 'pagePanels', 'localId'
    ]

    return remove_disposable_keys(response, disposable_keys)

class ServiceDeskManager:
    """
    A class to manage interactions with the Atlassian Service Desk API and to fetch
    request parameters necessary for creating service desk requests.

    Attributes
    ----------
    base_url : str
        The base URL for the Atlassian account.
    username : str
        The username for authentication.
    auth_token : str
        The authentication token or password.
    service_desk : ServiceDesk
```

```
        An instance of the Atlassian ServiceDesk client.
    jira : Jira
        An instance of the Atlassian Jira client.

    Methods
    -------
    get_service_desks() -> List[Dict]:
        Fetches and returns all service desk projects.
    get_request_types(portal_id: int) -> List[Dict]:
        Fetches and returns all request types for a specific service desk project.
    fetch_form(portal_id: int, request_type_id: int) -> Dict:
        Fetches the fields and parameters for the specified service desk request type.
    validate_field_data(portal_id: int, request_type_id: int, field_data: dict) -> bool:
        Validates the provided field data against the required fields from the request
parameters.
    create_service_desk_request(request_type: str, reporter_email: str,
                                field_data: dict, portal_id: str) -> Dict:
        Creates a service desk request with the specified parameters.
    """

    def __init__(self, base_url: str, username: str, auth_token: str):
        """
        Initializes the ServiceDeskManager class with authentication details.

        Parameters
        ----------
        base_url : str
            The base URL for the Atlassian account.
        username : str
            The username for authentication.
        auth_token : str
            The authentication token or password.
        """
        self.base_url = base_url
        self.username = username
        self.auth_token = auth_token
        self.auth_header = {
            "Authorization": f"Basic {base64.b64encode(f'{username}:
{auth_token}'.encode()).decode()}",
            "X-Atlassian-Token": "no-check"
        }
        self.default_headers = {
            "accept": "*/*",
            "content-type": "application/json",
            "x-requested-with": "XMLHttpRequest"
        }
        self.all_headers = {**self.default_headers, **self.auth_header}
        self.service_desk = ServiceDesk(url=base_url, username=username,
password=auth_token)
        self.jira = Jira(url=base_url, username=username, password=auth_token)

    def get_service_desks(self) -> List[Dict]:
        """
        Fetches and returns all service desk projects.

        Returns
        -------
        List[Dict]
            A list of dictionaries containing service desk project details.
        """
        return self.service_desk.get_service_desks()

    def get_request_types(self, service_desk_id: int, group_id: int) -> List[Dict]:
        """
        Fetches and returns all request types for a specific service desk project.

        Parameters
        ----------
        service_desk_id : int
            The ID of the service desk project.
        group_id : int
            The ID of the group.

        Returns
        -------
        List[Dict]
            A list of dictionaries containing request type details.
        """
        return self.service_desk.get_request_types(service_desk_id=service_desk_id,
group_id=group_id)

    def fetch_form(self, portal_id: int, request_type_id: int) -> Dict:
        """
        Fetches the fields and parameters for the specified service desk request type,
        including additional options for Proforma fields if they exist.

        Parameters
        ----------
        portal_id : int
            The ID of the service desk project (portalId).
        request_type_id : int
            The ID of the request type.
        Returns
        -------
        Dict
            The cleaned JSON response containing the fields, parameters, and additional
options.
        """
        form_data = self._fetch_form_data(portal_id, request_type_id)
        additional_options = self._fetch_proforma_options(portal_id, request_type_id)
        autocomplete_options = self._fetch_autocomplete_options(form_data)
        form_data['reqCreate']['proformaTemplateForm']["proformaFieldOptions"] =
additional_options
        form_data['reqCreate']["autocompleteOptions"] = autocomplete_options
        return form_data

    def _fetch_form_data(self, portal_id: int, request_type_id: int) -> Dict:
        """
        Fetches the form data for the specified service desk request type.

        Parameters
        ----------
        portal_id : int
            The ID of the service desk project (portalId).
        request_type_id : int
            The ID of the request type.

        Returns
        -------
        Dict
            The JSON response containing the fields and parameters.
        """
        url = f"{self.base_url}/rest/servicedesk/1/customer/models"
        headers = self.all_headers
        body = {
            "options": {
                "portalWebFragments": {
                    "portalId": portal_id,
                    "requestTypeId": request_type_id,
                    "portalPage": "CREATE_REQUEST"
                },
                "portal": {"id": portal_id},
                "reqCreate": {"portalId": portal_id, "id": request_type_id},
                "portalId": portal_id
            },
            "models": ["portalWebFragments", "portal", "reqCreate"],
            "context": {
                "helpCenterAri": "ari:cloud:help::help-center/023eca6c-913d-41af-
a182-61e86fd72ccc/de1070f9-b9dd-460c-b02f-104fc367db40",
                "clientBasePath": f"{self.base_url}/servicedesk/customer"
            }
        }
        response = requests.post(url, headers=headers, json=body)
        response.raise_for_status()

        return clean_response({**response.json(), "portalId": portal_id, "requestTypeId":
request_type_id})

    def _fetch_proforma_options(self, portal_id: int, request_type_id: int) -> Dict:
        """
        Fetches additional options for Proforma fields using a separate API endpoint.

        Parameters
        ----------
        portal_id : int
            The ID of the service desk project (portalId).
        request_type_id : int
            The ID of the request type.

        Returns
        -------
        Dict
            A dictionary containing the additional Proforma field options.
        """
        try:
            headers = self.all_headers
            tenant_info_url = f"{self.base_url}/_edge/tenant_info"
            tenant_info_response = requests.get(tenant_info_url, headers=headers)
            tenant_info_response.raise_for_status()
            cloud_id = tenant_info_response.json()["cloudId"]

            form_choices_url = f"{self.base_url}/gateway/api/proforma/portal/cloudid/
{cloud_id}/api/3/portal/{portal_id}/requesttype/{request_type_id}/formchoices"
            form_choices_response = requests.get(form_choices_url, headers=headers)
            form_choices_response.raise_for_status()
        except requests.exceptions.HTTPError as e:
            return {}
        return form_choices_response.json()

    def _fetch_autocomplete_options(self, form_data: dict) -> List[Dict[Any, Any]]:
        """
        Fetches autocomplete options for a specific field in a service desk request form.
        Parameters
        ----------
        portal_id : int
            The ID of the service desk project (portalId).
        request_id : int
            The ID of the specific service desk request.
        customfield_id : str
            The ID of the custom field for which autocomplete options are being fetched.

        Returns
        -------
        Dict
            A dictionary containing the autocomplete options for the specified custom field.
        """
        # TODO: discover how to paginate the request, since there is a hasNextPage field
in the response
        portal_id = form_data["portalId"]
        request_id = form_data["reqCreate"]["id"]
        field_map = {"fieldValueMap": {}, "query": ""}
        field_map_values = {field["fieldId"]: "" for field in form_data['reqCreate']['fields']}
        field_map["fieldValueMap"] = field_map_values
        autocomplete_fields = [field for field in form_data["reqCreate"]["fields"] if
field.get("autoCompleteUrl", "") and field.get("fieldType") != "organisationpicker"]
        additional_options = []
        for field in autocomplete_fields:
            customfield_id = field["fieldId"]
            try:
                headers = self.all_headers
                autocomplete_url = f"{self.base_url}/rest/servicedesk/cmdb/1/customer/portal/
{portal_id}/request/{request_id}/field/{customfield_id}/autocomplete"
                response = requests.post(autocomplete_url, headers=headers,
json=field_map)
                response.raise_for_status()
                response_dict = response.json()
            except requests.exceptions.HTTPError as e:
                print(f"Failed to fetch autocomplete options for field {customfield_id}: {e}")
                response_dict = {}
            response_dict = {
                **response_dict,
                "fieldId": customfield_id,
                "fieldType": field.get("fieldType", ""),
                "fieldLabel": field.get("label", ""),
                "fieldDescription": field.get("description", ""),
                "fieldRequired": field.get("required", ""),
                "fieldDisplayed": field.get("displayed", ""),
                "fieldPresetValues": field.get("presetValues", ""),
            }
            additional_options.append(response_dict)
        return additional_options


    def create_request(self, form_filled: ServiceDeskFormFilled) -> Dict:
        """
        Creates a service desk request with the specified parameters.

        Parameters
        ----------
        form_filled : ServiceDeskFormFilled
            An instance of ServiceDeskFormFilled containing validated user input.
        reporter_email : str
            The email of the reporter.

        Returns
        -------
        Dict
            The response from the API after creating the request.
        """
        field_data = form_filled.to_request_payload()

        portal_id = form_filled.form.service_desk_id
        request_type_id = form_filled.form.request_type_id

        url = f"{self.base_url}/servicedesk/customer/portal/{portal_id}/create/
{request_type_id}"
        headers = {
            "Content-Type": "application/x-www-form-urlencoded",
            "Accept": "application/json",
            **self.auth_header
        }

        params = requests.models.RequestEncodingMixin._encode_params(field_data)
        response = requests.post(url, headers=headers, data=params)

        if response.status_code in (201, 200):
            return response.json()
        else:
            raise ServiceDeskRequestError(response.status_code, response.text)import
pytest
from unittest.mock import Mock, patch
from your_module import ServiceDeskFormClient, FORM_DIDNT_FETCH_ERROR
from service_desk_manager import ServiceDeskManager
from form_parser import ServiceDeskFormParser
from form_manager import ServiceDeskFormManager
from response import CreateRequestResponseParser

@pytest.fixture
def client():
    return ServiceDeskFormClient("https://example.atlassian.net", "username", "token")

@pytest.fixture
def mock_service_desk_manager():
    with patch('your_module.ServiceDeskManager') as mock:
        yield mock

@pytest.fixture
def mock_form_parser():
    with patch('your_module.ServiceDeskFormParser') as mock:
        yield mock

@pytest.fixture
def mock_form_manager():
    with patch('your_module.ServiceDeskFormManager') as mock:
        yield mock

@pytest.fixture
def mock_response_parser():
    with patch('your_module.CreateRequestResponseParser') as mock:
        yield mock

def test_init(client):
    assert isinstance(client.service_desk_manager, ServiceDeskManager)
    assert client.form_manager is None

def test_fetch_and_parse_form(client, mock_service_desk_manager,
mock_form_parser, mock_form_manager):
    mock_form = {'some': 'data'}
    mock_service_desk_manager.return_value.fetch_form.return_value = mock_form
    mock_form_obj = Mock()
    mock_form_parser.parse.return_value = mock_form_obj

    client.fetch_and_parse_form(1, 2)

    mock_service_desk_manager.return_value.fetch_form.assert_called_once_with(port
al_id=1, request_type_id=2)
    mock_form_parser.parse.assert_called_once_with(mock_form)
    mock_form_manager.assert_called_once_with(mock_form_obj)
    assert isinstance(client.form_manager, Mock)
```

```
def test_list_fields_without_fetch(client):
    with pytest.raises(ValueError, match=FORM_DIDNT_FETCH_ERROR):
        client.list_fields()

def test_list_fields_with_fetch(client):
    client.form_manager = Mock()
    client.list_fields()
    client.form_manager.list_fields.assert_called_once()

def test_list_field_values_without_fetch(client):
    with pytest.raises(ValueError, match=FORM_DIDNT_FETCH_ERROR):
        client.list_field_values("field_name")

def test_list_field_values_with_fetch(client):
    client.form_manager = Mock()
    client.list_field_values("field_name")
    client.form_manager.list_field_values.assert_called_once_with("field_name")

def test_set_form_values_without_fetch(client):
    with pytest.raises(ValueError, match=FORM_DIDNT_FETCH_ERROR):
        client.set_form_values({"field": "value"})

def test_set_form_values_with_fetch(client):
    client.form_manager = Mock()
    client.form_manager.set_field_values.return_value = {"filled": "form"}
    result = client.set_form_values({"field": "value"})
    client.form_manager.set_field_values.assert_called_once_with({"field": "value"})
    assert result == {"filled": "form"}

def test_create_request(client, mock_response_parser):
    client.service_desk_manager.create_request = Mock()
    client.service_desk_manager.create_request.return_value = {"response": "data"}
    mock_response_parser.parse.return_value = Mock()

    filled_values = {"field": "value"}
    result = client.create_request(filled_values)

    client.service_desk_manager.create_request.assert_called_once_with(filled_values)
    mock_response_parser.parse.assert_called_once_with({"response": "data"})
    assert isinstance(result, Mock)

@pytest.mark.parametrize("method,args", [
    ("list_fields", []),
    ("list_field_values", ["field_name"]),
    ("set_form_values", [{"field": "value"}]),
])
```

```
def test_methods_raise_error_without_fetch(client, method, args):
    with pytest.raises(ValueError, match=FORM_DIDNT_FETCH_ERROR):
        getattr(client, method)(*args)

if __name__ == "__main__":
    pytest.main()import pytest
from datetime import datetime
from typing import Dict, Any
from form_parser import ServiceDeskForm, ServiceDeskFormField,
ServiceDeskFormFieldValue
from your_module import ServiceDeskFormValidator  # Replace 'your_module' with the
actual module name

@pytest.fixture
def sample_form():
    return ServiceDeskForm(
        id="FORM-1",
        service_desk_id="SD-1",
        request_type_id="RT-1",
        project_id="PROJ-1",
        portal_name="Test Portal",
        portal_description="Test Description",
        form_name="Test Form",
        form_description_html="<p>Test Form Description</p>",
        fields=[
            ServiceDeskFormField(
                field_type="dt",
                field_id="date_field",
                label="Date Field",
                required=True,
                displayed=True
            ),
            ServiceDeskFormField(
                field_type="select",
                field_id="select_field",
                label="Select Field",
                required=True,
                displayed=True,
                values=[
                    ServiceDeskFormFieldValue(value="option1", label="Option 1"),
                    ServiceDeskFormFieldValue(value="option2", label="Option 2")
                ]
            ),
            ServiceDeskFormField(
                field_type="text",
                field_id="text_field",
                label="Text Field",
                required=True,
                displayed=True
            ),
            ServiceDeskFormField(
                field_type="adf",
                field_id="adf_field",
                label="ADF Field",
                required=True,
                displayed=True
            ),
            ServiceDeskFormField(
                field_type="cascadingselect",
                field_id="cascading_field",
                label="Cascading Field",
                required=True,
                displayed=True,
                values=[
                    ServiceDeskFormFieldValue(
                        value="main1",
                        label="Main 1",
                        children=[
                            ServiceDeskFormFieldValue(value="sub1", label="Sub 1"),
                            ServiceDeskFormFieldValue(value="sub2", label="Sub 2")
                        ]
                    ),
                    ServiceDeskFormFieldValue(
                        value="main2",
                        label="Main 2",
                        children=[
                            ServiceDeskFormFieldValue(value="sub3", label="Sub 3"),
                            ServiceDeskFormFieldValue(value="sub4", label="Sub 4")
                        ]
                    )
                ]
            )
        ]
    )

@pytest.fixture
def validator():
    return ServiceDeskFormValidator()

def test_validate_valid_form(sample_form, validator):
    filled_values = {
        "date_field": "2023-05-15T10:30",
        "select_field": "option1",
        "text_field": "Sample text",
        "adf_field": '{"type": "doc", "content": [{"type": "paragraph", "content": [{"type": "text",
"text": "Sample ADF"}]}]}',
        "cascading_field": "main1",
        "cascading_field:1": "sub1"
    }
    validator.validate(filled_values, sample_form)  # Should not raise any exception

def test_validate_invalid_date(sample_form, validator):
    filled_values = {"date_field": "invalid-date"}
    with pytest.raises(ValueError, match="Invalid date-time value"):
        validator.validate(filled_values, sample_form)

def test_validate_invalid_select(sample_form, validator):
    filled_values = {"select_field": "invalid-option"}
    with pytest.raises(ValueError, match="Invalid choice value"):
        validator.validate(filled_values, sample_form)

def test_validate_invalid_text(sample_form, validator):
    filled_values = {"text_field": 12345}  # Not a string
    with pytest.raises(ValueError, match="Invalid text value"):
        validator.validate(filled_values, sample_form)

def test_validate_invalid_adf(sample_form, validator):
    filled_values = {"adf_field": '{"invalid": "json"}'}
    with pytest.raises(ValueError, match="Invalid ADF value"):
        validator.validate(filled_values, sample_form)

def test_validate_missing_field(sample_form, validator):
    filled_values = {"non_existent_field": "value"}
    with pytest.raises(ValueError, match="Field 'non_existent_field' not found in the
form"):
        validator.validate(filled_values, sample_form)

def test_validate_cascading_select_valid(sample_form, validator):
    filled_values = {
        "cascading_field": "main1",
        "cascading_field:1": "sub1"
    }
    validator.validate(filled_values, sample_form)  # Should not raise any exception

def test_validate_cascading_select_invalid_main(sample_form, validator):
    filled_values = {
        "cascading_field": "invalid_main",
        "cascading_field:1": "sub1"
    }
    with pytest.raises(ValueError, match="Invalid main field value"):
        validator.validate(filled_values, sample_form)

def test_validate_cascading_select_invalid_sub(sample_form, validator):
    filled_values = {
        "cascading_field": "main1",
        "cascading_field:1": "invalid_sub"
    }
    with pytest.raises(ValueError, match="Invalid subfield value"):
        validator.validate(filled_values, sample_form)

def test_validate_cascading_select_missing_main(sample_form, validator):
    filled_values = {
        "cascading_field:1": "sub1"
    }
    with pytest.raises(ValueError, match="Main field .* is not set"):
        validator.validate(filled_values, sample_form)

def test_validate_dt_field(sample_form, validator):
    assert validator._validate_dt("2023-05-15T10:30") == True
    assert validator._validate_dt("invalid-date") == False

def test_validate_choice_field(sample_form, validator):
    choices = ["option1", "option2"]
    assert validator._validate_choice("option1", choices) == True
    assert validator._validate_choice("invalid", choices) == False

def test_validate_text_field(sample_form, validator):
    assert validator._validate_text("Valid text") == True
    assert validator._validate_text(12345) == False

def test_validate_adf_field(sample_form, validator):
    valid_adf = '{"type": "doc", "content": [{"type": "paragraph", "content": [{"type": "text",
"text": "Valid ADF"}]}]}'
    invalid_adf = '{"invalid": "json"}'
    assert validator._validate_adf(valid_adf) == True
    assert validator._validate_adf(invalid_adf) == False

def test_get_field_by_id_or_label(sample_form, validator):
    assert validator._get_field_by_id_or_label(sample_form, "date_field") is not None
    assert validator._get_field_by_id_or_label(sample_form, "Date Field") is not None
    assert validator._get_field_by_id_or_label(sample_form, "non_existent") is None

def test_get_value_by_label_or_id(sample_form, validator):
    values = sample_form.fields[1].values  # select_field values
    assert validator._get_value_by_label_or_id(values, "option1") is not None
    assert validator._get_value_by_label_or_id(values, "Option 1") is not None
    assert validator._get_value_by_label_or_id(values, "non_existent") is None

if __name__ == "__main__":
    pytest.main()import pytest
from typing import Dict, Any
from your_module import ServiceDeskFormManager, ServiceDeskFormFilled,
ServiceDeskForm, ServiceDeskFormField, ServiceDeskFormFieldValue
from field_validator import ServiceDeskFormValidator

@pytest.fixture
def sample_form():
    return ServiceDeskForm(
        id="FORM-1",
        service_desk_id="SD-1",
        request_type_id="RT-1",
        project_id="PROJ-1",
        portal_name="Test Portal",
        portal_description="Test Description",
        form_name="Test Form",
        form_description_html="<p>Test Form Description</p>",
        template_id=123,
        template_form_uuid="UUID-456",
        fields=[
            ServiceDeskFormField(
                field_type="text",
                field_id="summary",
                label="Summary",
                required=True,
                displayed=True
            ),
            ServiceDeskFormField(
                field_type="select",
                field_id="priority",
                label="Priority",
                required=True,
                displayed=True,
                values=[
                    ServiceDeskFormFieldValue(value="high", label="High"),
                    ServiceDeskFormFieldValue(value="medium", label="Medium"),
                    ServiceDeskFormFieldValue(value="low", label="Low")
                ]
            ),
            ServiceDeskFormField(
                field_type="cascadingselect",
                field_id="category",
                label="Category",
                required=True,
                displayed=True,
                values=[
                    ServiceDeskFormFieldValue(
                        value="hardware",
                        label="Hardware",
                        children=[
                            ServiceDeskFormFieldValue(value="laptop", label="Laptop"),
                            ServiceDeskFormFieldValue(value="desktop", label="Desktop")
                        ]
                    ),
                    ServiceDeskFormFieldValue(
                        value="software",
                        label="Software",
                        children=[
                            ServiceDeskFormFieldValue(value="os", label="Operating System"),
                            ServiceDeskFormFieldValue(value="application", label="Application")
                        ]
                    )
                ]
            ),
            ServiceDeskFormField(
                field_type="rt",
                field_id="description",
                label="Description",
                required=True,
                displayed=True,
                is_proforma_field=True,
                proforma_question_id="PROFORMA-1"
            )
        ]
    )

@pytest.fixture
def form_manager(sample_form):
    return ServiceDeskFormManager(sample_form)

def test_list_fields(form_manager):
    fields = form_manager.list_fields()
    assert len(fields) == 4
    assert fields[0]["label"] == "Summary"
    assert fields[1]["id"] == "priority"
    assert fields[2]["type"] == "cascadingselect"
    assert fields[3]["description"] == ""
```

```
def test_list_field_values(form_manager):
    priority_values = form_manager.list_field_values("priority")
    assert len(priority_values) == 3
    assert priority_values[0]["label"] == "High"
    assert priority_values[1]["value"] == "medium"

def test_list_field_values_with_parent(form_manager):
    category_values = form_manager.list_field_values("category", "Hardware")
    assert len(category_values) == 2
    assert category_values[0]["label"] == "Laptop"
    assert category_values[1]["value"] == "desktop"

def test_list_field_values_invalid_field(form_manager):
    with pytest.raises(ValueError):
        form_manager.list_field_values("invalid_field")

def test_list_field_values_invalid_parent(form_manager):
    with pytest.raises(ValueError):
        form_manager.list_field_values("category", "Invalid Parent")

def test_validate_valid_form(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "high",
        "category": ("hardware", "laptop"),
        "description": "Test description"
    }
    assert form_manager.validate(filled_values) == True

def test_validate_missing_required_field(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "high",
        "category": ("hardware", "laptop")
    }
    with pytest.raises(ValueError, match="Missing required fields"):
        form_manager.validate(filled_values)

def test_validate_invalid_choice(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "invalid_priority",
        "category": ("hardware", "laptop"),
        "description": "Test description"
    }
    with pytest.raises(ValueError, match="Invalid choice value"):
        form_manager.validate(filled_values)

def test_set_field_values_valid(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "high",
        "category": ("hardware", "laptop"),
        "description": "Test description"
    }
    form_filled = form_manager.set_field_values(filled_values)
    assert isinstance(form_filled, ServiceDeskFormFilled)
    assert form_filled.filled_values["summary"] == "Test summary"
    assert form_filled.filled_values["priority"] == "high"
    assert form_filled.filled_values["category"] == "hardware"
    assert form_filled.filled_values["category:1"] == "laptop"
    assert form_filled.filled_values["description"] == "Test description"

def test_set_field_values_with_labels(form_manager):
    filled_values = {
        "Summary": "Test summary",
        "Priority": "High",
        "Category": ("Hardware", "Laptop"),
        "Description": "Test description"
    }
    form_filled = form_manager.set_field_values(filled_values)
    assert form_filled.filled_values["summary"] == "Test summary"
    assert form_filled.filled_values["priority"] == "high"
    assert form_filled.filled_values["category"] == "hardware"
    assert form_filled.filled_values["category:1"] == "laptop"
    assert form_filled.filled_values["description"] == "Test description"

def test_set_field_values_invalid(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "invalid_priority",
        "category": ("hardware", "laptop"),
        "description": "Test description"
    }
    with pytest.raises(ValueError):
        form_manager.set_field_values(filled_values)

def test_to_request_payload(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "high",
        "category": ("hardware", "laptop"),
        "description": "Test description"
    }
    form_filled = form_manager.set_field_values(filled_values)
    payload = form_filled.to_request_payload()

    assert "summary=Test+summary" in payload
    assert "priority=high" in payload
    assert "category=hardware" in payload
    assert "category%3A1=laptop" in payload
    assert "projectId=PROJ-1" in payload
    assert "proformaFormData=%7B" in payload  # Start of JSON-encoded
proformaFormData

    # Decode the proformaFormData to check its contents
    import urllib.parse
    import json
    decoded_payload = urllib.parse.unquote(payload)
    proforma_data_start = decoded_payload.index('proformaFormData=') +
len('proformaFormData=')
    proforma_data_json = decoded_payload[proforma_data_start:]
    proforma_data = json.loads(proforma_data_json)

    assert proforma_data["templateFormId"] == 123
    assert "PROFORMA-1" in proforma_data["answers"]
    assert proforma_data["answers"]["PROFORMA-1"]["adf"]["content"][0]["content"][0]
["text"] == "Test description"

def test_create_request_payload(form_manager):
    filled_values = {
        "summary": "Test summary",
        "priority": "high",
        "category": ("hardware", "laptop"),
        "description": "Test description"
    }
    payload = form_manager.create_request_payload(filled_values)

    assert payload["summary"] == "Test summary"
    assert payload["priority"] == "high"
    assert payload["category"] == "hardware"
    assert payload["category:1"] == "laptop"
    assert "proformaFormData" in payload

    proforma_data = json.loads(payload["proformaFormData"])
    assert proforma_data["templateFormId"] == 123
    assert "PROFORMA-1" in proforma_data["answers"]
    assert proforma_data["answers"]["PROFORMA-1"] == "Test description"

def test_convert_labels_to_ids(form_manager):
    filled_values = {
        "Summary": "Test summary",
        "Priority": "High",
        "Category": ("Hardware", "Laptop"),
        "Description": "Test description"
    }
    converted = form_manager._convert_labels_to_ids(filled_values)
    assert converted["summary"] == "Test summary"
    assert converted["priority"] == "high"
    assert converted["category"] == "hardware"
    assert converted["category:1"] == "laptop"
    assert converted["description"] == "Test description"

def test_convert_labels_to_ids_invalid_field(form_manager):
    filled_values = {
        "Invalid Field": "Test value"
    }
    with pytest.raises(ValueError, match="Field 'Invalid Field' not found in the form"):
        form_manager._convert_labels_to_ids(filled_values)

def test_convert_labels_to_ids_invalid_value(form_manager):
    filled_values = {
        "Priority": "Invalid Priority"
    }
    with pytest.raises(ValueError, match="Invalid value 'Invalid Priority' for field"):
        form_manager._convert_labels_to_ids(filled_values)

if __name__ == "__main__":
    pytest.main()import pytest
from typing import Dict, Any
from your_sdk_module import ServiceDeskFormParser, ServiceDeskForm,
ServiceDeskFormField, ServiceDeskFormFieldValue

@pytest.fixture
def sample_json_data() -> Dict[str, Any]:
    return {
        "portal": {
            "id": "PORTAL-123",
            "name": "IT Help Desk",
            "description": "IT support portal",
            "serviceDeskId": "SD-456",
            "projectId": "PROJ-789"
        },
        "reqCreate": {
            "id": "REQ-001",
            "form": {
                "name": "IT Support Request",
                "descriptionHtml": "<p>Submit your IT support request</p>"
            },
            "fields": [
                {
                    "fieldType": "text",
                    "fieldId": "summary",
                    "label": "Summary",
                    "description": "Brief description of the issue",
                    "descriptionHtml": "<p>Brief description of the issue</p>",
                    "required": True,
                    "displayed": True
                },
                {
                    "fieldType": "textarea",
                    "fieldId": "description",
                    "label": "Description",
                    "description": "Detailed description of the issue",
                    "descriptionHtml": "<p>Detailed description of the issue</p>",
                    "required": True,
                    "displayed": True,
                    "rendererType": "wiki"
                },
                {
                    "fieldType": "select",
                    "fieldId": "priority",
                    "label": "Priority",
                    "description": "Issue priority",
                    "descriptionHtml": "<p>Issue priority</p>",
                    "required": True,
                    "displayed": True,
                    "values": [
                        {"value": "high", "label": "High", "selected": False},
                        {"value": "medium", "label": "Medium", "selected": True},
                        {"value": "low", "label": "Low", "selected": False}
                    ]
                }
            ],
            "proformaTemplateForm": {
                "updated": "2023-09-15T12:00:00Z",
                "design": {
                    "settings": {
                        "templateId": 123,
                        "templateFormUuid": "FORM-UUID-456"
                    },
                    "questions": {
                        "CUSTOM-001": {
                            "type": "text",
                            "label": "Custom Field",
                            "description": "A custom field",
                            "validation": {"rq": True},
                            "jiraField": "customfield_10001"
                        }
                    }
                },
                "proformaFieldOptions": {
                    "fields": {
                        "customfield_10001": [
                            {"id": "option1", "label": "Option 1"},
                            {"id": "option2", "label": "Option 2"}
                        ]
                    }
                }
            }
        },
        "xsrfToken": "TOKEN-789"
    }

    def test_parse_service_desk_form(sample_json_data):
        form = ServiceDeskFormParser.parse(sample_json_data)

        assert isinstance(form, ServiceDeskForm)
        assert form.id == "PORTAL-123"
        assert form.service_desk_id == "SD-456"
        assert form.request_type_id == "REQ-001"
        assert form.project_id == "PROJ-789"
        assert form.portal_name == "IT Help Desk"
        assert form.portal_description == "IT support portal"
        assert form.form_name == "IT Support Request"
        assert form.form_description_html == "<p>Submit your IT support request</p>"
        assert form.updated_at == "2023-09-15T12:00:00Z"
        assert form.template_id == 123
        assert form.template_form_uuid == "FORM-UUID-456"
        assert form.atl_token == "TOKEN-789"

    def test_parse_standard_fields(sample_json_data):
        form = ServiceDeskFormParser.parse(sample_json_data)

        assert len(form.fields) == 4  # 3 standard fields + 1 proforma field
        summary_field = next(field for field in form.fields if field.field_id == "summary")
        assert summary_field.field_type == "text"
        assert summary_field.label == "Summary"
        assert summary_field.required == True

        description_field = next(field for field in form.fields if field.field_id == "description")
        assert description_field.field_type == "textarea"
        assert description_field.renderer_type == "wiki"

        priority_field = next(field for field in form.fields if field.field_id == "priority")
        assert priority_field.field_type == "select"
        assert len(priority_field.values) == 3
        assert priority_field.values[1].value == "medium"
        assert priority_field.values[1].selected == True

    def test_parse_proforma_field(sample_json_data):
        form = ServiceDeskFormParser.parse(sample_json_data)

        proforma_field = next(field for field in form.fields if field.is_proforma_field)
        assert proforma_field.field_type == "text"
        assert proforma_field.field_id == "customfield_10001"
        assert proforma_field.label == "Custom Field"
        assert proforma_field.required == True
        assert proforma_field.proforma_question_id == "CUSTOM-001"
        assert len(proforma_field.values) == 2
        assert proforma_field.values[0].value == "option1"
        assert proforma_field.values[1].label == "Option 2"

    def test_parse_cascadingselect_field():
        cascading_json = {
        "portal": {"id": "PORTAL-123", "name": "Test Portal", "serviceDeskId": "SD-456",
    "projectId": "PROJ-789"},
        "reqCreate": {
            "id": "REQ-001",
            "form": {"name": "Test Form", "descriptionHtml": ""},
            "fields": [
                {
                    "fieldType": "cascadingselect",
                    "fieldId": "cascading",
                    "label": "Cascading Select",
                    "required": True,
                    "displayed": True,
                    "values": [
                        {
                            "value": "parent1",
                            "label": "Parent 1",
                            "children": [
                                {"value": "child1", "label": "Child 1"},
                                {"value": "child2", "label": "Child 2"}
                            ]
                        },
                        {
                            "value": "parent2",
                            "label": "Parent 2",
                            "children": [
                                {"value": "child3", "label": "Child 3"}
                            ]
                        }
                    ]
                }
            ]
        }
        }

        form = ServiceDeskFormParser.parse(cascading_json)

        assert len(form.fields) == 2  # Main field + subfield

        main_field = form.fields[0]
        assert main_field.field_type == "cascadingselect"
        assert main_field.field_id == "cascading"
        assert len(main_field.values) == 2
        assert len(main_field.values[0].children) == 2
        assert len(main_field.values[1].children) == 1

        subfield = form.fields[1]
        assert subfield.field_id == "cascading:1"
        assert subfield.label == "Cascading Select (Subfield)"
        assert subfield.depends_on == "cascading"

    def test_parse_autocomplete_field():
        autocomplete_json = {
        "portal": {"id": "PORTAL-123", "name": "Test Portal", "serviceDeskId": "SD-456",
    "projectId": "PROJ-789"},
        "reqCreate": {
            "id": "REQ-001",
            "form": {"name": "Test Form", "descriptionHtml": ""},
            "fields": [
                {
                    "fieldType": "cmdbobjectpicker",
                    "fieldId": "cmdb",
                    "label": "CMDB Object",
                    "required": True,
                    "displayed": True,
                    "autoCompleteUrl": "/api/autocomplete"
                }
            ],
            "autocompleteOptions": [
                {
                    "fieldId": "cmdb",
                    "results": [
                        {
                            "objectId": "OBJ-001",
                            "label": "Server 1",
                            "workspaceId": "WS-001",
                            "objectKey": "SERVER-1",
                            "objectType": {
                                "objectTypeId": "TYPE-001",
                                "id": "server",
                                "name": "Server",
                                "description": "Server object"
                            },
                            "attributes": [
                                {
                                    "objectTypeAttributeId": "ATTR-001",
                                    "objectTypeAttribute": {
                                        "name": "Hostname",
                                        "type": "string",
                                        "description": "Server hostname"
                                    },
                                    "objectAttributeValues": ["server1.example.com"]
                                }
                            ]
                        }
                    ]
                }
            ]
        }
        }

        form = ServiceDeskFormParser.parse(autocomplete_json)

        assert len(form.fields) == 1

        cmdb_field = form.fields[0]
        assert cmdb_field.field_type == "cmdbobjectpicker"
        assert cmdb_field.field_id == "cmdb"
```

```
    assert cmdb_field.auto_complete_url == "/api/autocomplete"
    assert len(cmdb_field.values) == 1

    value = cmdb_field.values[0]
    assert value.value == "OBJ-001"
    assert value.label == "Server 1"
    assert value.additional_data["workspaceId"] == "WS-001"
    assert value.additional_data["objectKey"] == "SERVER-1"
    assert value.additional_data["objectType"]["name"] == "Server"
    assert value.additional_data["objectTypeAttributeName"] == "Hostname"
    assert value.additional_data["objectTypeAttributeValues"] == ["server1.example.com"]

if __name__ == "__main__":
    pytest.main()import pytest
from your_module import CreateRequestResponseParser, CreateRequestResponse,
Reporter, Issue, IssueField

@pytest.fixture
def sample_response_data():
    return {
        "reporter": {
            "email": "john.doe@example.com",
            "displayName": "John Doe",
            "avatarUrl": "https://example.com/avatar.jpg",
            "accountId": "12345"
        },
        "requestTypeName": "IT Support",
        "key": "IT-123",
        "issueType": "10000",
        "issueTypeName": "Service Request",
        "issue": {
            "id": 123,
            "key": "IT-123",
            "reporter": {
                "email": "john.doe@example.com",
                "displayName": "John Doe",
                "avatarUrl": "https://example.com/avatar.jpg",
                "accountId": "12345"
            },
            "participants": ["user1", "user2"],
            "organisations": ["org1", "org2"],
            "sequence": 1,
            "serviceDeskKey": "SD-1",
            "requestTypeName": "IT Support",
            "requestTypeId": 10,
            "summary": "Need help with printer",
            "isNew": True,
            "status": "Open",
            "date": "2023-05-15T10:30:00Z",
            "friendlyDate": "Today",
            "fields": [
                {
                    "id": "summary",
                    "label": "Summary",
                    "value": {"text": "Need help with printer"}
                },
                {
                    "id": "description",
                    "label": "Description",
                    "value": {"text": "Printer not responding"}
                }
            ],
            "activityStream": ["Activity 1", "Activity 2"],
            "requestIcon": 1,
            "iconUrl": "https://example.com/icon.png",
            "canBrowse": True,
            "canAttach": True,
            "categoryKey": "printer",
            "creatorAccountId": "12345",
            "formKey": "FORM-1"
        },
        "canCreateIssues": True,
        "canAddComment": True,
        "issueLinkUrl": "https://example.com/issues/IT-123",
        "requestDetailsBaseUrl": "https://example.com/requests/"
    }

def test_parse_reporter():
    reporter_data = {
        "email": "john.doe@example.com",
        "displayName": "John Doe",
        "avatarUrl": "https://example.com/avatar.jpg",
        "accountId": "12345"
    }
    reporter = CreateRequestResponseParser._parse_reporter(reporter_data)
    assert isinstance(reporter, Reporter)
    assert reporter.email == "john.doe@example.com"
    assert reporter.display_name == "John Doe"
    assert reporter.avatar_url == "https://example.com/avatar.jpg"
    assert reporter.account_id == "12345"

def test_parse_issue_field():
    field_data = {
        "id": "summary",
        "label": "Summary",
        "value": {"text": "Need help with printer"}
    }
    issue_field = CreateRequestResponseParser._parse_issue_field(field_data)
    assert isinstance(issue_field, IssueField)
    assert issue_field.id == "summary"
    assert issue_field.label == "Summary"
    assert issue_field.value == {"text": "Need help with printer"}

def test_parse_issue(sample_response_data):
    issue_data = sample_response_data["issue"]
    issue = CreateRequestResponseParser._parse_issue(issue_data)
    assert isinstance(issue, Issue)
    assert issue.id == 123
    assert issue.key == "IT-123"
    assert isinstance(issue.reporter, Reporter)
    assert issue.participants == ["user1", "user2"]
    assert issue.organisations == ["org1", "org2"]
    assert issue.sequence == 1
    assert issue.service_desk_key == "SD-1"
    assert issue.request_type_name == "IT Support"
    assert issue.request_type_id == 10
    assert issue.summary == "Need help with printer"
    assert issue.is_new is True
    assert issue.status == "Open"
    assert issue.date == "2023-05-15T10:30:00Z"
    assert issue.friendly_date == "Today"
    assert len(issue.fields) == 2
    assert all(isinstance(field, IssueField) for field in issue.fields)
    assert issue.activity_stream == ["Activity 1", "Activity 2"]
    assert issue.request_icon == 1
    assert issue.icon_url == "https://example.com/icon.png"
    assert issue.can_browse is True
    assert issue.can_attach is True
    assert issue.category_key == "printer"
    assert issue.creator_account_id == "12345"
    assert issue.form_key == "FORM-1"

def test_parse_create_request_response(sample_response_data):
    response = CreateRequestResponseParser.parse(sample_response_data)
    assert isinstance(response, CreateRequestResponse)
    assert isinstance(response.reporter, Reporter)
    assert response.request_type_name == "IT Support"
    assert response.key == "IT-123"
    assert response.issue_type == "10000"
    assert response.issue_type_name == "Service Request"
    assert isinstance(response.issue, Issue)
    assert response.can_create_issues is True
    assert response.can_add_comment is True
    assert response.issue_link_url == "https://example.com/issues/IT-123"
    assert response.request_details_base_url == "https://example.com/requests/"

def test_parse_with_missing_optional_fields():
    minimal_data = {
        "reporter": {
            "email": "john.doe@example.com",
            "displayName": "John Doe",
            "avatarUrl": "https://example.com/avatar.jpg",
            "accountId": "12345"
        },
        "issue": {
            "id": 123,
            "key": "IT-123",
            "reporter": {
                "email": "john.doe@example.com",
                "displayName": "John Doe",
                "avatarUrl": "https://example.com/avatar.jpg",
                "accountId": "12345"
            },
            "summary": "Minimal issue"
        }
    }
    response = CreateRequestResponseParser.parse(minimal_data)
    assert isinstance(response, CreateRequestResponse)
    assert response.request_type_name == ""
    assert response.can_create_issues is False
    assert response.can_add_comment is False
    assert response.issue_link_url == ""
    assert response.request_details_base_url == ""
    assert response.issue.participants == []
    assert response.issue.organisations == []
    assert response.issue.sequence == 0
    assert response.issue.fields == []

def test_parse_with_empty_fields():
    data_with_empty_fields = {
        "reporter": {},
        "issue": {
            "reporter": {},
            "fields": [
                {"id": "empty_field"},
                {"label": "Empty Label"},
                {"value": {}}
            ]
        }
    }
    response = CreateRequestResponseParser.parse(data_with_empty_fields)
    assert isinstance(response, CreateRequestResponse)
    assert response.reporter.email == ""
    assert response.reporter.display_name == ""
    assert response.reporter.avatar_url == ""
    assert response.reporter.account_id == ""
    assert len(response.issue.fields) == 3
    assert response.issue.fields[0].id == "empty_field"
    assert response.issue.fields[0].label == ""
    assert response.issue.fields[0].value == {}
    assert response.issue.fields[1].id == ""
    assert response.issue.fields[1].label == "Empty Label"
    assert response.issue.fields[1].value == {}
    assert response.issue.fields[2].id == ""
    assert response.issue.fields[2].label == ""
    assert response.issue.fields[2].value == {}

if __name__ == "__main__":
    pytest.main()import pytest
import requests
import json
from unittest.mock import Mock, patch
from your_module import ServiceDeskManager, ServiceDeskRequestError,
ServiceDeskFormFilled, ServiceDeskForm

@pytest.fixture
def service_desk_manager():
    return ServiceDeskManager("https://example.atlassian.net", "username", "token")

@pytest.fixture
def mock_response():
    mock = Mock()
    mock.raise_for_status = Mock()
    return mock

@patch('requests.post')
@patch('requests.get')
def test_fetch_form(mock_get, mock_post, service_desk_manager, mock_response):
    # Mock the main form data response
    mock_post.return_value = mock_response
    mock_response.json.return_value = {
        "reqCreate": {
            "fields": [
                {"fieldId": "summary", "fieldType": "text"},
                {"fieldId": "description", "fieldType": "textarea"}
            ]
        }
    }

    # Mock the tenant info response
    mock_get.return_value = mock_response
    mock_response.json.side_effect = [
        {"cloudId": "CLOUD-123"},  # Tenant info response
        {},  # Proforma options response (empty in this case)
    ]

    form_data = service_desk_manager.fetch_form(1, 2)

    assert "reqCreate" in form_data
    assert "fields" in form_data["reqCreate"]
    assert len(form_data["reqCreate"]["fields"]) == 2
    assert form_data["reqCreate"]["fields"][0]["fieldId"] == "summary"
    assert "proformaFieldOptions" in form_data["reqCreate"]["proformaTemplateForm"]
    assert "autocompleteOptions" in form_data["reqCreate"]

@patch('requests.get')
def test_get_service_desks(mock_get, service_desk_manager):
    mock_get.return_value.json.return_value = {"values": [{"id": 1, "projectName": "IT
Help Desk"}]}

    service_desks = service_desk_manager.get_service_desks()

    assert len(service_desks) == 1
    assert service_desks[0]["id"] == 1
    assert service_desks[0]["projectName"] == "IT Help Desk"

@patch('requests.get')
def test_get_request_types(mock_get, service_desk_manager):
    mock_get.return_value.json.return_value = {"values": [{"id": 10, "name": "IT
Support"}]}

    request_types = service_desk_manager.get_request_types(1, 1)

    assert len(request_types) == 1
    assert request_types[0]["id"] == 10
    assert request_types[0]["name"] == "IT Support"
```

```
@patch('requests.post')
def test_create_request_success(mock_post, service_desk_manager):
    mock_response = Mock()
    mock_response.status_code = 201
    mock_response.json.return_value = {"issueKey": "SD-123"}
    mock_post.return_value = mock_response

    form = ServiceDeskForm(
        id="FORM-1",
        service_desk_id="1",
        request_type_id="10",
        project_id="PROJ-1",
        portal_name="Test Portal",
        form_name="Test Form",
        portal_description="Test Description",
        form_description_html="<p>Test Form Description</p>",
    )
    form_filled = ServiceDeskFormFilled(form=form, filled_values={"summary": "Test
Issue"})

    result = service_desk_manager.create_request(form_filled)

    assert result["issueKey"] == "SD-123"

@patch('requests.post')
def test_create_request_failure(mock_post, service_desk_manager):
    mock_response = Mock()
    mock_response.status_code = 400
    mock_response.text = "Bad Request"
    mock_post.return_value = mock_response

    form = ServiceDeskForm(
        id="FORM-1",
        service_desk_id="1",
        request_type_id="10",
        project_id="PROJ-1",
        portal_name="Test Portal",
        form_name="Test Form",
        portal_description="Test Description",
        form_description_html="<p>Test Form Description</p>",
    )
    form_filled = ServiceDeskFormFilled(form=form, filled_values={"summary": "Test
Issue"})

    with pytest.raises(ServiceDeskRequestError) as excinfo:
        service_desk_manager.create_request(form_filled)

    assert "400" in str(excinfo.value)
    assert "Bad Request" in str(excinfo.value)

def test_remove_disposable_keys():
    from your_module import remove_disposable_keys

    test_data = {
        "keep": "value",
        "remove": "value",
        "nested": {
            "keep": "value",
            "remove": "value"
        },
        "list": [
            {"keep": "value", "remove": "value"},
            {"keep": "value", "remove": "value"}
        ]
    }
    disposable_keys = ["remove"]

    result = remove_disposable_keys(test_data, disposable_keys)

    assert "keep" in result
    assert "remove" not in result
    assert "keep" in result["nested"]
    assert "remove" not in result["nested"]
    assert "keep" in result["list"][0]
    assert "remove" not in result["list"][0]

@patch('requests.post')
def test_fetch_autocomplete_options(mock_post, service_desk_manager):
    mock_response = Mock()
    mock_response.json.return_value = {
        "results": [
            {"objectId": "OBJ-1", "label": "Option 1"},
            {"objectId": "OBJ-2", "label": "Option 2"}
        ]
    }
    mock_post.return_value = mock_response

    form_data = {
        "portalId": 1,
        "reqCreate": {
            "id": 10,
            "fields": [
                {
                    "fieldId": "customfield_10000",
                    "fieldType": "cmdbobjectpicker",
                    "autoCompleteUrl": "/autocomplete",
                    "label": "CMDB Object",
                    "description": "Select a CMDB object",
                    "required": True,
                    "displayed": True
                }
            ]
        }
    }

    result = service_desk_manager._fetch_autocomplete_options(form_data)

    assert len(result) == 1
    assert result[0]["fieldId"] == "customfield_10000"
    assert result[0]["fieldType"] == "cmdbobjectpicker"
    assert result[0]["fieldLabel"] == "CMDB Object"
    assert len(result[0]["results"]) == 2
    assert result[0]["results"][0]["objectId"] == "OBJ-1"

if __name__ == "__main__":
    pytest.main()
```