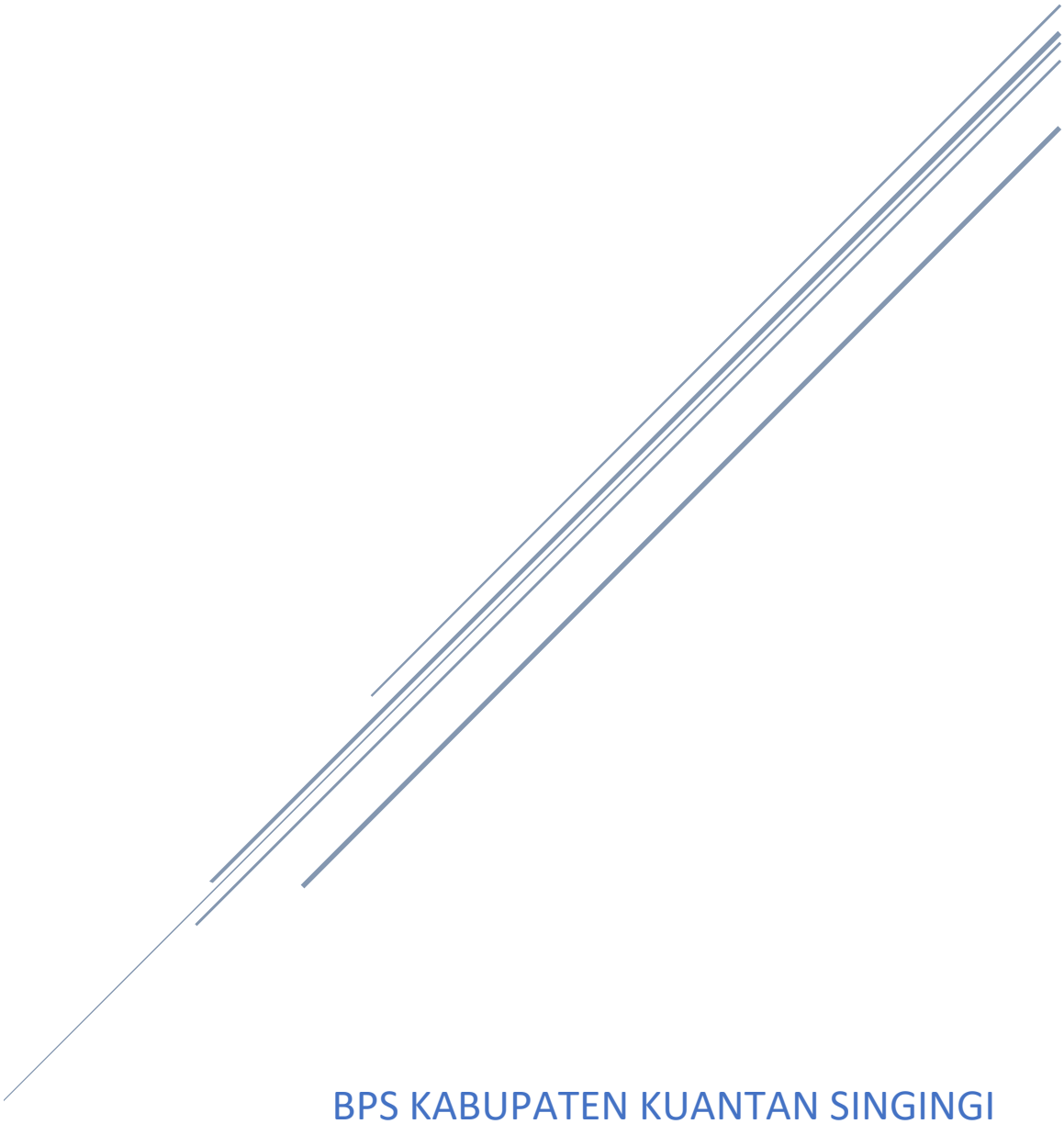
III.A.11. DOKUMEN RULE VALIDASI

Sistem Informasi Penilaian Pegawai Terbaik (SIPIA)



BPS KABUPATEN KUANTAN SINGINGI

2021

III.A.11 RULE VALIDASI SISTEM INFORMASI PENILAIAN PEGAWAI TERBAIK (SIPIA)

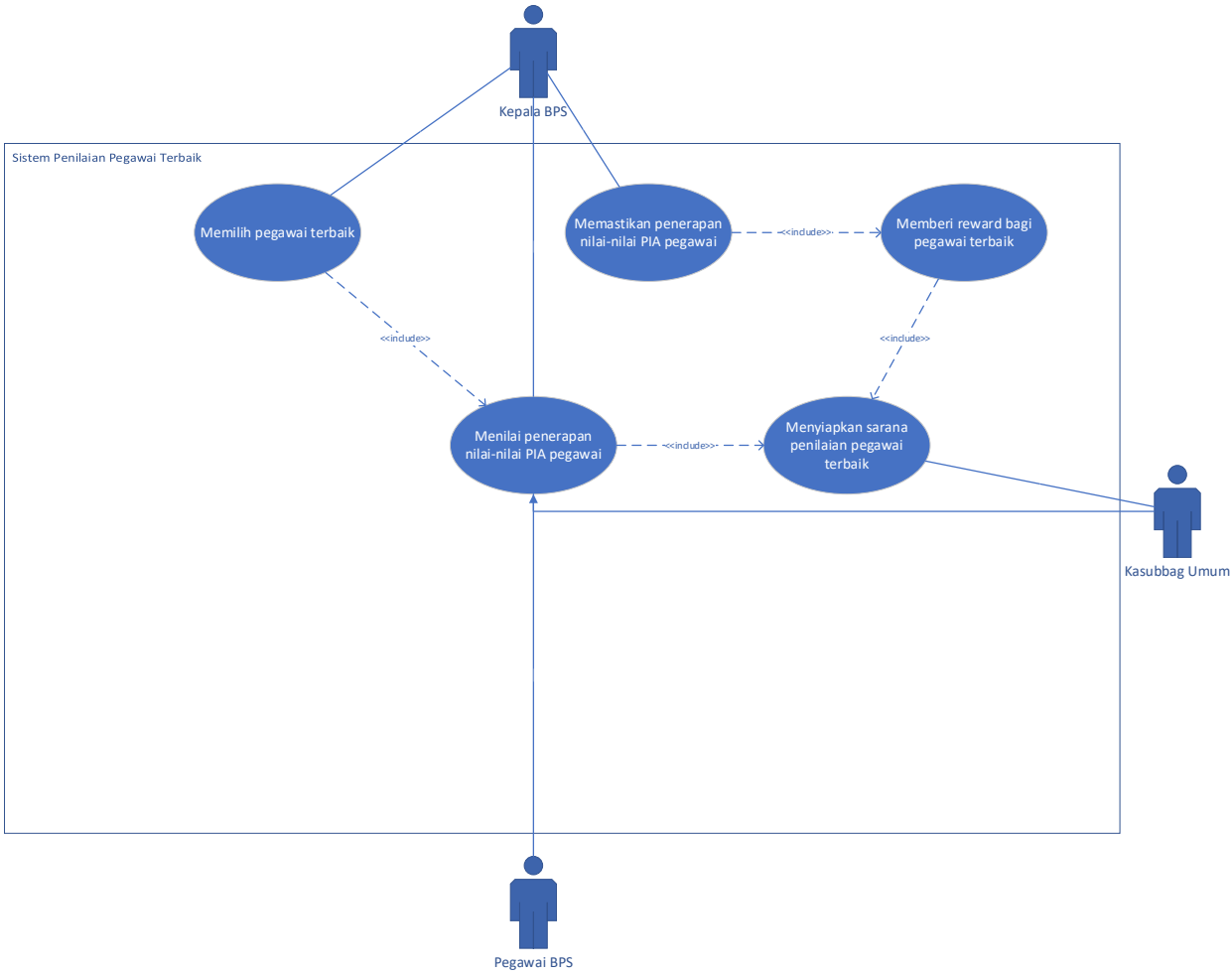
A. Deskripsi Singkat SIstem

Sistem Informasi Penilaian Pegawai Terbaik (SIPIA) adalah sistem penilaian pegawai terbaik dengan cara menilai penerapan nilai-nilai PIA pegawai berbasis *web*. SIPIA memiliki fitur-fitur sebagai berikut:

- Form penilaian untuk seluruh pegawai di-*generate* secara otomatis untuk setiap periodenya.
- Form penilaian terdiri dari penilaian profesionalitas, integritas, dan keamanahan pegawai.
- Hanya pengguna yang telah masuk kedalam sistem yang dapat mengakses form penilaian.
- Memiliki tampilan status penilaian pegawai terhadap pegawai lain apakah sudah atau belum lengkap.
- Memiliki fitur untuk memantau status kelengkapan penilaian seluruh pegawai.
- Dapat menampilkan hasil penilaian sementara secara *realtime*.
- Dapat menyimpan *track record* penilaian pegawai kedalam *database*.

Penjelasan dari gambar 1 adalah sebagai berikut:

1. Form penilaian untuk seluruh pegawai di-*generate* secara otomatis oleh sistem untuk setiap periodenya.
2. Pegawai wajib menilai seluruh pegawai kecuali dirinya sendiri.
3. Kasubbag umum memberikan informasi *progress* penilaian ke Kepala BPS.
4. Kepala BPS memilih pegawai terbaik dari 5 kandidat pegawai dengan nilai tertinggi.



Gambar 1. Gambaran umum SIPIA

SIPIA termasuk dalam sistem aplikasi kompleks karena memilki lebih dari 5 subsistem, diantaranya login, home, session, CRUD (*create*, *read*, *update*, *delete*) data, dan monitoring.

B. Daftar *Rule* Validasi

Login dan Registrasi

- *Field username* dan password tidak boleh kosong.
- Atribut username panjang maksimal 150 karakter dan harus unik.
- Atribut password minimal 8 karakter, tidak boleh telalu mirip dengan informasi pribadi, tidak boleh angka semua. dam tidak boleh berupa sandi yang umum digunakan, seperti abcd, 123456, dll.

```python
username = models.CharField(
        _("username"),
        max_length=150,
        unique=True,
        help_text=_(
            "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only."
        ),
        validators=[username_validator],
        error_messages={
            "unique": _("A user with that username already exists."),
        },
    )
```

Gambar 2. Rule validasi *username*

```python
def validate_password(password, user=None, password_validators=None):
    """
    Validate that the password meets all validator requirements.

    If the password is valid, return ``None``.
    If the password is invalid, raise ValidationError with all error messages.
    """
    errors = []
    if password_validators is None:
        password_validators = get_default_password_validators()
    for validator in password_validators:
        try:
            validator.validate(password, user)
        except ValidationError as error:
            errors.append(error)
    if errors:
        raise ValidationError(errors)


def password_changed(password, user=None, password_validators=None):
    """
    Inform all validators that have implemented a password_changed() method
    that the password has been changed.
    """
    if password_validators is None:
        password_validators = get_default_password_validators()
    for validator in password_validators:
```

```
        password_changed = getattr(validator, "password_changed", lambda *a:
None)
        password_changed(password, user)


def password_validators_help_texts(password_validators=None):
    """
    Return a list of all help texts of all configured validators.
    """
    help_texts = []
    if password_validators is None:
        password_validators = get_default_password_validators()
    for validator in password_validators:
        help_texts.append(validator.get_help_text())
    return help_texts


def _password_validators_help_text_html(password_validators=None):
    """
    Return an HTML string with all help texts of all configured validators
    in an <ul>.
    """
    help_texts = password_validators_help_texts(password_validators)
    help_items = format_html_join(
        "", "<li>{}</li>", ((help_text,) for help_text in help_texts)
    )
    return format_html("<ul>{}</ul>", help_items) if help_items else ""


password_validators_help_text_html = lazy(_password_validators_help_text_html,
str)


class MinimumLengthValidator:
    """
    Validate that the password is of a minimum length.
    """

    def __init__(self, min_length=8):
        self.min_length = min_length

    def validate(self, password, user=None):
        if len(password) < self.min_length:
            raise ValidationError(
                ngettext(
                    "This password is too short. It must contain at least "
```

```
                    "%(min_length)d character.",
                    "This password is too short. It must contain at least "
                    "%(min_length)d characters.",
                    self.min_length,
                ),
                code="password_too_short",
                params={"min_length": self.min_length},
            )

    def get_help_text(self):
        return ngettext(
            "Your password must contain at least %(min_length)d character.",
            "Your password must contain at least %(min_length)d characters.",
            self.min_length,
        ) % {"min_length": self.min_length}


def exceeds_maximum_length_ratio(password, max_similarity, value):
    """
    Test that value is within a reasonable range of password.

    The following ratio calculations are based on testing SequenceMatcher like
    this:

    for i in range(0,6):
      print(10**i, SequenceMatcher(a='A', b='A'*(10**i)).quick_ratio())

    which yields:

    1 1.0
    10 0.18181818181818182
    100 0.019801980198019802
    1000 0.001998001998001998
    10000 0.00019998000199980003
    100000 1.999980000199998e-05

    This means a length_ratio of 10 should never yield a similarity higher than
    0.2, for 100 this is down to 0.02 and for 1000 it is 0.002. This can be
    calculated via 2 / length_ratio. As a result we avoid the potentially
    expensive sequence matching.
    """
    pwd_len = len(password)
    length_bound_similarity = max_similarity / 2 * pwd_len
    value_len = len(value)
    return pwd_len >= 10 * value_len and value_len < length_bound_similarity
```

```
class UserAttributeSimilarityValidator:
    """
    Validate that the password is sufficiently different from the user's
    attributes.

    If no specific attributes are provided, look at a sensible list of
    defaults. Attributes that don't exist are ignored. Comparison is made to
    not only the full attribute value, but also its components, so that, for
    example, a password is validated against either part of an email address,
    as well as the full address.
    """

    DEFAULT_USER_ATTRIBUTES = ("username", "first_name", "last_name", "email")

    def __init__(self, user_attributes=DEFAULT_USER_ATTRIBUTES,
max_similarity=0.7):
        self.user_attributes = user_attributes
        if max_similarity < 0.1:
            raise ValueError("max_similarity must be at least 0.1")
        self.max_similarity = max_similarity

    def validate(self, password, user=None):
        if not user:
            return

        password = password.lower()
        for attribute_name in self.user_attributes:
            value = getattr(user, attribute_name, None)
            if not value or not isinstance(value, str):
                continue
            value_lower = value.lower()
            value_parts = re.split(r"\W+", value_lower) + [value_lower]
            for value_part in value_parts:
                if exceeds_maximum_length_ratio(
                    password, self.max_similarity, value_part
                ):
                    continue
                if (
                    SequenceMatcher(a=password, b=value_part).quick_ratio()
                    >= self.max_similarity
                ):
                    try:
                        verbose_name = str(
```

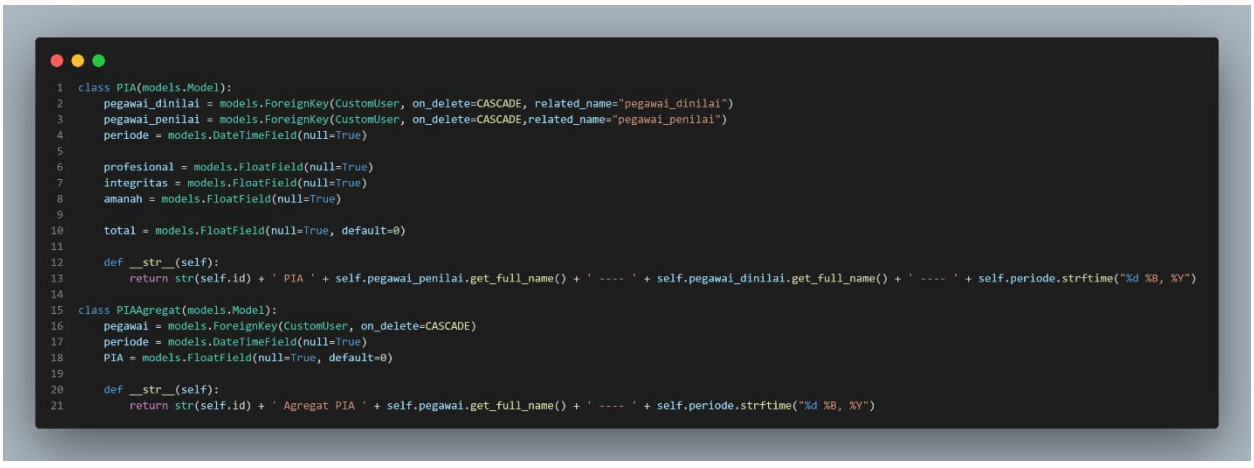
```
                            user._meta.get_field(attribute_name).verbose_name
                        )
                    except FieldDoesNotExist:
                        verbose_name = attribute_name
                    raise ValidationError(
                        _("The password is too similar to the
%(verbose_name)s."),
                        code="password_too_similar",
                        params={"verbose_name": verbose_name},
                    )

    def get_help_text(self):
        return _(
            "Your password can’t be too similar to your other personal
information."
        )


class CommonPasswordValidator:
    """
    Validate that the password is not a common password.

    The password is rejected if it occurs in a provided list of passwords,
    which may be gzipped. The list Django ships with contains 20000 common
    passwords (lowercased and deduplicated), created by Royce Williams:
    https://gist.github.com/roycewilliams/281ce539915a947a23db17137d91aeb7
    The password list must be lowercased to match the comparison in validate().
    """

    @cached_property
    def DEFAULT_PASSWORD_LIST_PATH(self):
        return Path(__file__).resolve().parent / "common-passwords.txt.gz"

    def __init__(self, password_list_path=DEFAULT_PASSWORD_LIST_PATH):
        if password_list_path is
CommonPasswordValidator.DEFAULT_PASSWORD_LIST_PATH:
            password_list_path = self.DEFAULT_PASSWORD_LIST_PATH
        try:
            with gzip.open(password_list_path, "rt", encoding="utf-8") as f:
                self.passwords = {x.strip() for x in f}
        except OSError:
            with open(password_list_path) as f:
                self.passwords = {x.strip() for x in f}

    def validate(self, password, user=None):
```

```
        if password.lower().strip() in self.passwords:
            raise ValidationError(
                _("This password is too common."),
                code="password_too_common",
            )

    def get_help_text(self):
        return _("Your password can’t be a commonly used password.")


class NumericPasswordValidator:
    """
    Validate that the password is not entirely numeric.
    """

    def validate(self, password, user=None):
        if password.isdigit():
            raise ValidationError(
                _("This password is entirely numeric."),
                code="password_entirely_numeric",
            )

    def get_help_text(self):
        return _("Your password can’t be entirely numeric.")
```

Gambar 3. Rule validasi *password*

Input Penilaian Kegiatan

- Atribut profesional, integritas hanya dapat diisi dengan angka.



```
class PIA(models.Model):
    pegawai_dinilai = models.ForeignKey(CustomUser, on_delete=CASCADE, related_name="pegawai_dinilai")
    pegawai_penilai = models.ForeignKey(CustomUser, on_delete=CASCADE,related_name="pegawai_penilai")
    periode = models.DateTimeField(null=True)

    profesional = models.FloatField(null=True)
    integritas = models.FloatField(null=True)
    amanah = models.FloatField(null=True)

    total = models.FloatField(null=True, default=0)

    def __str__(self):
        return str(self.id) + ' PIA ' + self.pegawai_penilai.get_full_name() + ' ---- ' + self.pegawai_dinilai.get_full_name() + ' ---- ' + self.periode.strftime("%d %B, %Y")

class PIAAgregat(models.Model):
    pegawai = models.ForeignKey(CustomUser, on_delete=CASCADE)
    periode = models.DateTimeField(null=True)
    PIA = models.FloatField(null=True, default=0)

    def __str__(self):
        return str(self.id) + ' Agregat PIA ' + self.pegawai.get_full_name() + ' ---- ' + self.periode.strftime("%d %B, %Y")
```

Gambar 4. Rule validasi entri penilaian PIA