Issue #3291

→ Pontoon string completed if in any of 3 states:

① Approved (reviewed & accepted by translator)
② Pretranslated (machine-translated & accepted)
③ Has warnings (accepted but with non-blocking issues)

completed = approved + pretranslated + warnings

complete = BOOLEAN on whole locale/project meaning all strings are done

completed_strings = count of individual strings in done state

② Pretranslated

pretranslation task (pretranslation / tasks.py : 33)
Celery background job triggered after sync
ProjectLocale.pretranslation_enabled = True;

for any locale-project pairs

for each untranslated entity found, call get_pretranslation()
① Translation memory first - 100% match in TranslationMemoryEntry for said locale
② Google Translate fallback if locale has google_translate_code

creates Translation record with flags:

```
user = pt_authors[author_key],   # bot account
approved = False,
pretranslated = True,            # acceptance set by system
active = True
```

2 AUTHORS, either "Google Translate" bot OR "Translation Memory" bot

- pretranslated can promote to active (starts counting) state
  complete → needs human to sign off
  completed_strings → if good enough, can go

fuzzy: needs review

Count	Condition
approved	approved = True ∧ no errors ∧ no warnings
pretranslated	pretranslated = True ∧ no errors ∧ no warnings
errors	(approved || pretranslated || fuzzy) ∧ errors
warnings	(approved || pretranslated || fuzzy) ∧ warnings
unreviewed	!approved, !rejected, !pretranslated, !fuzzy

- completed: has accepted translation (approved || pretranslated || fuzzy) ∧ !errors

- for any given (entity, locale) can have many Translation rows but only one can have active = True, current (shown in UI string list, stats count computed against that)

- pretranslation setting active = True says is string representing translation right now, not just candidate in history

Database Table) TranslatedResource

- each row is one file (resource) in one locale, row has real integer columns

- cached counters, stored

  approved_strings
  pretranslated_strings
  strings_with_errors
  strings_with_warnings
  unreviewed_strings
  total_strings

- can sum up given column across all TranslatedResource rows for a language and can get aggregated statistics

- everything in stats system built on top of stored counters

4 Layers that are Changed

(1) AggregatedStats (Python mixin) /aggregated_stats.py

- mixin) class that doesn't stand alone, exists to be mixed into another class, adding methods / properties to it
- shared logic once in mixin, multiple classes inherit from it

- AggregatedStats mixed into 3 model classes) Locale, Project, ProjectLocale
  all 3 objects get some computed properties automatically

- _stats property) access any stat on Locale instance calls _stats

```
@cached_property
def _stats(self) -> dict[str, int]:
    return self.aggregated_stats_query.string_stats(
        count_disabled=True, count_system_projects=True
    )
```

defined on Locale model itself, returns TranslatedResource queryset filtered to specific locale

```python
@cached_property
def _stats(self) -> dict[str, int]:
    return self.aggregated_stats_query.string_stats(
        count_disabled=True, count_system_projects=True
    )
```

→ runs SQL SUM, returns python dict approved: 15, pretranslated: 0 etc. etc.

→ run SQL query first time, cache result in memory, never hit database again for subsequent stat access on same object

- each stat then is a python property reading from cached dictionary
- derived stats computed from stored counters in python

```
51
52+     @property
53+     def completed_strings(self) -> int:
54+         return (
55+             self.approved_strings
56+             + self.pretranslated_strings
57+             + self.strings_with_warnings
58+         )
59+
60      @property
```

} derived stat that aren't stored in TranslationResource

When layer is used
+ single model instance

fire one SQL query (_stats) → cache results → compute answer in Python

time complexity) N + 1, 1 query to fetch locales +N queries to fetch each one's stats } SLOW!

(2) Queryset Methods stats_data()

locale.py
project.py
project_locale.py

QuerySet) object that represents database query which hasn't run yet
build chaining methods e.g.
.filter(), .annotate(), .order_by()

actual SQL fires when iterate over or call .all()

compound columns) multiple columns together as single logical unit for filtering, ordering, grouping etc. .annotate()

stats_data()
- constructs one SQL query, fetches multiple objects, computes all stats ∀ object

annotates w/ completed, API field called completed_strings

annotation name unambiguous in SQL, verbose public name in serializer

creates approved, pretranslated, warnings etc.

references those

```python
def stats_data(self, project=None):
    if project is not None:
        query = self.filter(
            translatedresources__resource__project=project,
            translatedresources__resource__obsolete=False,
        )
    else:
        query = self.filter(
            translatedresources__resource__project__disabled=False,
            translatedresources__resource__project__system_project=False,
            translatedresources__resource__project__visibility="public",
            translatedresources__resource__obsolete=False,
        )

    return query.annotate(
        total=Sum("translatedresources__total_strings", default=0),
        approved=Sum("translatedresources__approved_strings", default=0),
        pretranslated=Sum("translatedresources__pretranslated_strings", default=0),
        errors=Sum("translatedresources__strings_with_errors", default=0),
        warnings=Sum("translatedresources__strings_with_warnings", default=0),
        unreviewed=Sum("translatedresources__unreviewed_strings", default=0),
    ).annotate(
        missing=F("total")
        - F("approved")
        - F("pretranslated")
        - F("errors")
        - F("warnings"),
        completed=F("approved") + F("pretranslated") + F("warnings"),
        is_complete=Case(
            When(
                total=F("approved") + F("warnings"),
                then=Value(True),
            ),
            default=Value(False),
            output_field=BooleanField(),
        ),
    )
```

→ sums values across every TranslatedResource row belonging to that locale

→ every queryset result has extra attributes attached to it due to the summing of the values

→ referring to already computed values (happens inside SQL before any data sent to python)

(3) Serializer /serializers.py

→ takes python object (model instance / queryset row) converts to python dict

DRF encodes as JSON sends back as API response

```python
TRANSLATION_STATS_FIELDS = [
    "total_strings",
    "approved_strings",
    "pretranslated_strings",
    "strings_with_warnings",
    "strings_with_errors",
    "missing_strings",
    "completed_strings",
    "unreviewed_strings",
    "complete",
]
```

→ python list for DRF to ensure field should appear in JSON output of every serializer that uses list

work w/ annotated queryset objects, see (2)

```python
# DO NOT REMOVE serializers.SerializerMetaclass, it is required for serializer functionality
class TranslationStatsMixin(metaclass=serializers.SerializerMetaclass):
    total_strings = serializers.SerializerMethodField()
    approved_strings = serializers.SerializerMethodField()
    pretranslated_strings = serializers.SerializerMethodField()
    strings_with_warnings = serializers.SerializerMethodField()
    strings_with_errors = serializers.SerializerMethodField()
    missing_strings = serializers.SerializerMethodField()
    completed_strings = serializers.SerializerMethodField()
    unreviewed_strings = serializers.SerializerMethodField()
    complete = serializers.SerializerMethodField()
```

→ doesn't look up model field w/ name but call method on serializer class to get value

→ each stat as serializer method field

```python
    def get_strings_with_errors(self, obj):
        return obj.errors

    def get_missing_strings(self, obj):
        return obj.missing

    def get_completed_strings(self, obj):
        return obj.completed

    def get_unreviewed_strings(self, obj):
        return obj.unreviewed
```

→ annotated locale/project / project_locale object from stats_data()

→ refers to step #2 computation

Naming Translation

Layer	Name	Reason
DB annotation	`completed`	Short, used in SQL arithmetic
Serializer method	`get_completed_strings`	DRF convention: prefix method with `get_`
JSON output / field name	`completed_strings`	Verbose, unambiguous for API consumers

} difference in nomenclature also useful for programming nuances

```python

class LocaleSerializer(DynamicFieldsModelSerializer):
    class Meta:
        model = Locale
        fields = [
            "code",
            "name",
            "direction",
            "population",
            "cldr_plurals",
            "plural_rule",
            "script",
            "google_translate_code",
            "ms_terminology_code",
            "ms_translator_code",
            "systran_translate_code",
            "team_description",
        ] + TRANSLATION_STATS_FIELDS


class ProjectSerializer(DynamicFieldsModelSerializer):
    contact = serializers.SerializerMethodField()

    class Meta:
        model = Project
```

don't use TranslationStatsMixin

→ plain subclasses

- if no serializer in Meta.fields declared, falls back to obj. — — — as attribute

- since inherit from Agg. Stats. (1) triggers DB query per object (N+1) but only in base serializers

- only if base serializers used directly, if nested variants, annotated queryset used instead

(4) Tests

```python
    translated_resource.total_strings = 25
    translated_resource.approved_strings = 15
    translated_resource.pretranslated_strings = 0
    translated_resource.strings_with_errors = 3
    translated_resource.strings_with_warnings = 2
    translated_resource.missing_strings = 5
    translated_resource.unreviewed_strings = 5
    translated_resource.save()
```

Use predefined values to generate tests

completed = 17

aggregated (2 locales) = 34