UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE – UERN
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS – FANAT
DEPARTAMENTO DE INFORMÁTICA – DI

Pedro Belo de Almeida Neto

SIRAF-NE - Módulo de Gestão

MOSSORÓ / RN
2023

Pedro Belo de Almeida Neto

SIRAF-NE - Módulo de Gestão

Relatório apresentado ao curso de Ciência da Computação da Universidade do Estado do Rio Grande no Norte como requisito da disciplina de Trabalho de Diplomação, sob a orientação do Prof. Dr. Rommel Wladimir de Lima.

Mossoró / RN
2023

Pedro Belo de Almeida Neto

SIRAF-NE - Módulo de Gestão

Relatório apresentado como pré-requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade do Estado do Rio Grande do Norte – UERN, submetida à aprovação da banca examinadora composta pelos seguintes membros:

Aprovado em ____ / ____ / _____

Banca Examinadora

_____

Prof. Dr. Rommel Wladimir de Lima (Orientador)
Universidade do Estado do Rio Grande do Norte – UERN

_____

Profa. Me. Jessica Neiva de Figueiredo Leite (Examinador)
Universidade do Estado do Rio Grande do Norte – UERN

_____

Prof. Dr. Maximiliano Araujo da Silva Lopes (Examinador)
Universidade do Estado do Rio Grande do Norte – UERN

_____

Profa. Dra. Ceres Germanna Braga Morais (Examinador)
Universidade do Estado do Rio Grande do Norte – UERN

SUMÁRIO

1. INTRODUÇÃO

A Agricultura Familiar desempenha um papel vital do ponto de vista econômico e social, sendo a principal responsável pela disponibilização de alimentos para a sociedade brasileira. Ela permite aos agricultores familiares terem um meio de subsistência além da possibilidade do comércio de produções excedentes.

No Nordeste, segundo um artigo publicado no Boletim Regional, Urbano e Ambiental do IPEA (Instituto de Pesquisa Econômica Aplicada) (2021, p. 1), “[...] apesar dos efeitos de uma das maiores secas registradas em sua história recente, a agricultura familiar continua sendo a principal forma de produção e trabalho no campo no final da segunda década do século XXI, abrangendo 47,2% do total nacional. [...]”

Diante desse cenário, o Sistema de Informação Regional da Agricultura Familiar do Nordeste, sendo mencionado a partir desse momento apenas como SIRAF-NE, surgiu como uma forma de ampliar os resultados das políticas públicas referentes à Agricultura Familiar por meio da automação de uma série de processos realizados por cada Estado do Nordeste, centralizando as principais informações sobre os agricultores e suas produções.

2. OBJETIVOS

Objetivo Geral

Desenvolver um sistema para automação de processos relacionados à Agricultura Familiar realizados pelas Secretarias de cada Estado do Nordeste.

Objetivos Específicos

- Utilizar o modelo Cliente Servidor para implementar um sistema multinível com distribuição vertical.
- Utilizar tecnologias modernas e consolidadas no mercado para a implementação do sistema.

- Permitir um alto grau de desacoplamento de serviços importantes do ponto de vista de uma aplicação WEB, tais como banco de dados e arquivos estáticos.
- Definição da interface para a comunicação entre os módulos do SIRAF-NE. De modo que ela possibilite a separação entre os elementos de acesso público e privado de acordo com o perfil de cada usuário.

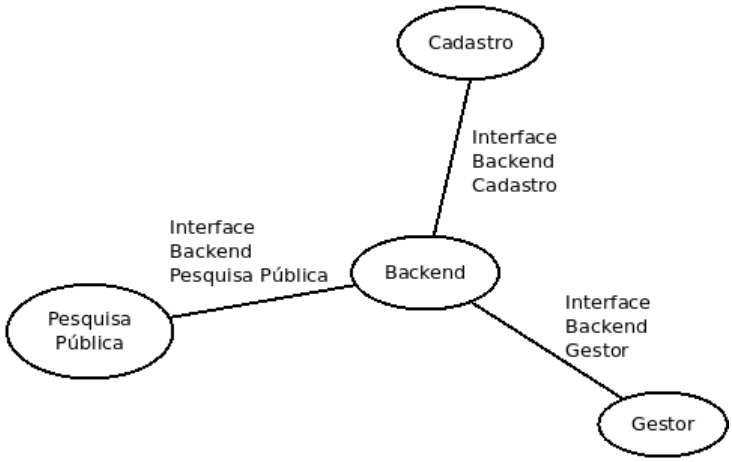
3. METODOLOGIA

Durante o desenvolvimento do sistema foi utilizado um processo de software incremental e iterativo baseado na metodologia ágil scrum. As reuniões envolvendo todos os membros da equipe foram realizadas em períodos semanais em encontros remotos, com duração aproximada de 30 minutos.

4. MÓDULOS DO SIRA-NE

O SIRAF-NÉ é composto por quatro módulos: Backend, Cadastro, Gestão e Pesquisa Pública. Cada um deles é implementado de forma independente, de modo que os módulos de Cadastro, Gestão e Pesquisa Pública se comunicam com o módulo Backend através de um conjunto de *endpoints* em que o intercâmbio de informações é realizado, na maioria dos casos, através do formato JSON por meio do protocolo HTTP.

Figura 1 - Visão Geral do SIRAF-NE



Fonte: autoria própria

O módulo backend é codificado utilizando a linguagem python com o *framework* Django e a biblioteca django-rest-framework. Django é um *framework* para desenvolvimento de aplicações *fullstack* e monolíticas com base no padrão MTV (Model, Template e View). Para permitir a criação de um sistema multinível foi utilizada a biblioteca django-rest-framework, permitindo, dessa forma, a separação do *backend* e *frontend*. Os módulos de Cadastro, Gestor e Pesquisa Pública são codificados utilizando a linguagem Javascript com o *framework* Vue JS. O Vue JS é um *framework* utilizado na criação de *single page applications*. Cada módulo construído utilizando o Vue JS utiliza como template base o coreui.

Backend

Controla a aplicação e validação das regras de negócio. Ele atua sendo a interface para o funcionamento apropriado de todos os módulos que compõem o SIRAF-NE. As interfaces são compostas pela API pública e privada. A API pública disponibiliza informações acerca dos principais indicadores referentes aos ofertantes individuais, grupos informais, associações e cooperativas. A API Privada também fornece indicadores, mas, com maior grau de detalhes e que é de uso exclusivo dos gestores de cada Estado que integra o SIRAF-NE.

Cadastro

Permite que agentes externos tenham acesso ao SIRAF-NE. Entre as principais funcionalidades desse módulo, estão os seguintes submódulos: gerenciamento de contas, cadastro de ofertante, cadastro de ofertas e pedido de homologação. O gerenciamento de conta permite a criação e manutenção dos dados cadastrais do usuário. O cadastro de ofertante admite diferentes tipos: Ofertante Individual, Grupo Informal, Associações e Cooperativas. A inserção dos produtos ofertados é realizada de acordo com os ofertantes previamente cadastrados e a lista de produtos que foram definidos pelos Estados que compõem o SIRAF-NE. O pedido de homologação é um recurso que permite ao usuário externo solicitar ao gestor a validação dos dados cadastrais referentes aos ofertantes e produtos ofertados.

Gestão

Permite aos gestores terem acesso ao SIRAF-NE. Entre as principais funcionalidades desse módulo, estão os seguintes submódulos: gerenciamento de contas, homologação, biblioteca virtual e vitrine privada. O gerenciamento de contas permite a atualização dos dados cadastrais da conta gestora, assim como de contas externas. O módulo de homologação permite que gestores validarem os dados cadastrais dos ofertantes, bem como suas ofertas por meio do pedido de homologação do ofertante, sendo gerado ao final do processo o certificado de homologação. A biblioteca virtual permite a criação, edição e publicação de documentos de diversos tipos (edital, legislação, etc.). A vitrine privada permite a visualização de dados de forma visual, através gráficos ou em formato textual, podendo também, serem exportados em formato de planilha.

Pesquisa Pública

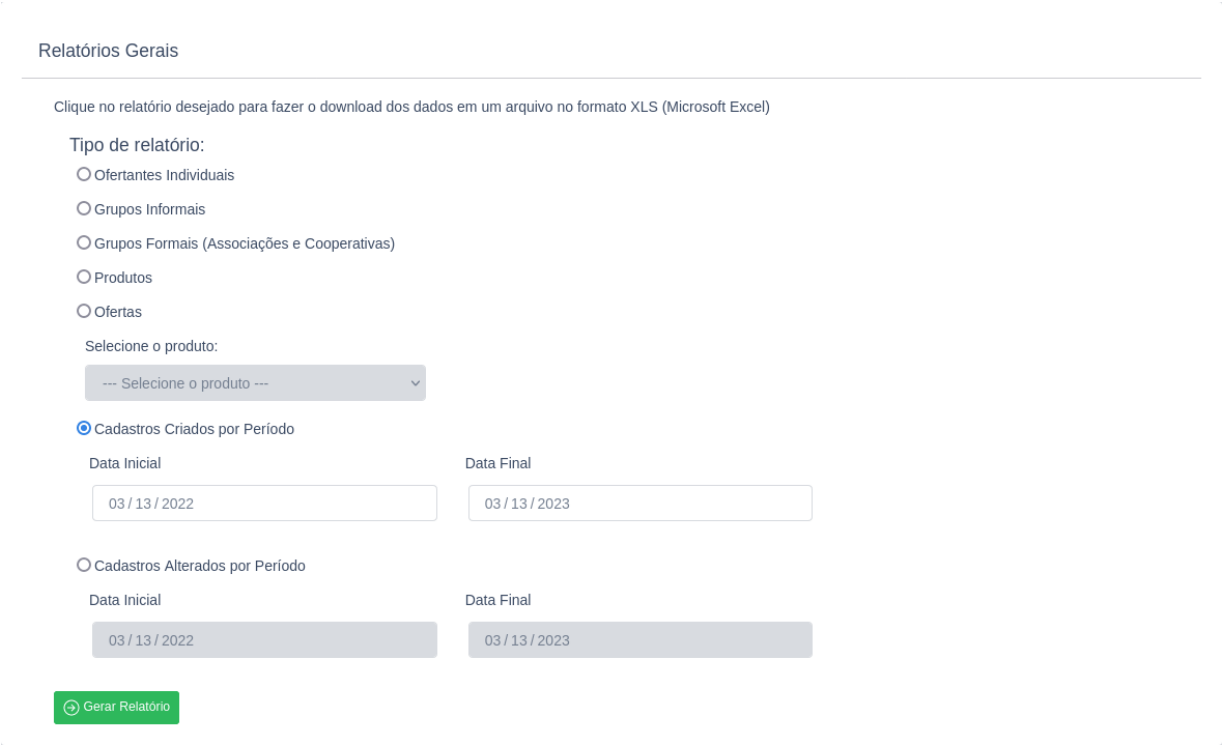
Permite aos visitantes terem acesso ao SIRAF-NE. Entre as principais funcionalidades desse módulo, estão os seguintes submódulos: banco de preços, biblioteca virtual, vitrine pública e validação do certificado de homologação. O banco de preços disponibiliza pesquisas de preço de mercado que serão utilizadas pelo Estado para aquisição dos diversos produtos da agricultura familiar. A biblioteca virtual concentra documentos de diversos tipos publicados por cada Estado do SIRAF-NE. A validação do certificado permite verificar se um determinado certificado de homologação é válido a partir de sua chave.

Considerando a diversidade de funcionalidades, mesmo levando em consideração apenas o módulo gestor, o foco do próximo tópico será direcionado apenas ao submódulo de gestor que atua na geração de relatórios.

5. GERAÇÃO DE RELATÓRIOS

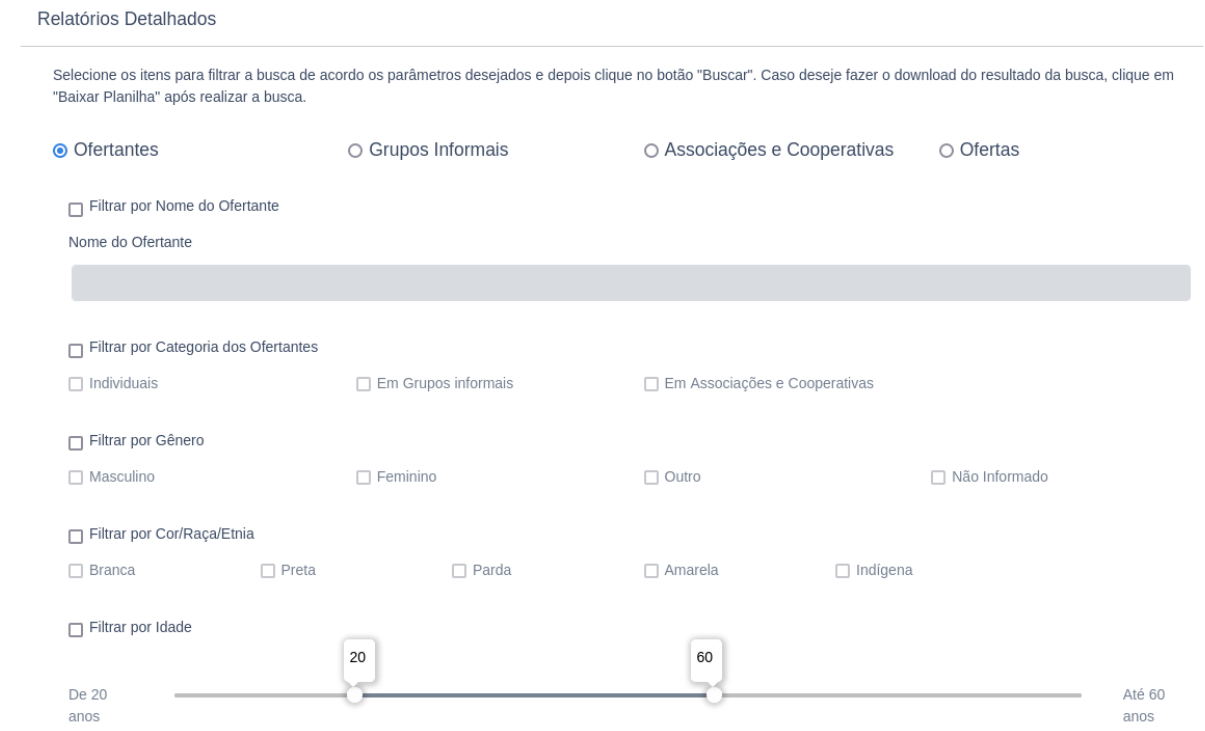
A geração de relatórios tem início com a ação “gerar relatório” definida no Módulo Gestor. Ela admite três opções: relatório quantitativo, relatório detalhado e relatório operacional, ilustrados nas figuras a seguir:

Figura 2 - Relatório Quantitativo



Fonte: autoria pŕopria

Figura 3 - Relatório detalhado



Fonte: autoria pŕopria

FIgura 4 - Relatório Operacional

Fonte: autoria pŕopria

Em seguida, o módulo gestor envia uma requisição HTTP ao Módulo Backend por meio de um dos seguintes *endpoints*:

- api/v1/reports/individual
- api/v1/reports/informal
- api/v1/reports/formal
- api/v1/reports/producer
- api/v1/reports/sirafne-monitoring
- api/v1/reports/sirafne-monitoring/user
- api/v1/reports/products
- api/v1/reports/group/formal/<int:pk>
- api/v1/reports/group/informal/<int:pk>
- api/v1/reports/offers/<int:product_id>
- api/v1/reports/group-and-producer-by-territory
- api/v1/reports/offer-by-territory

Ao receber a requisição, o servidor de aplicação repassa-a para o *framework* Django. Após isso, o *framework* faz o despacho da requisição para a *view* apropriada, de acordo com o *endpoint*.

O comportamento geral da *view* é criar uma instância de uma das subclasses da classe Serializer e, em seguida, executar o método *export* definido na classe Serializer.

Figura 5 - Classe bases para todas as outras views

```
class GenericExportView(generics.GenericAPIView):

    permission_classes = [permissions.IsAuthenticated]

    def post(self, request, **kwargs):

        self.user = request.user
        response = HttpResponse(content_type="application/vnd.ms-excel")

        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        report = serializer.export()

        filename = f"{serializer.report_title} - {str(datetime.now())}"

        response["Content-Disposition"] = f'attachment; filename="{filename}.xls"'
        report.save(response)
        return response
```

Fonte: autoria própria

Existem situações onde é importante definir se o serializer está tratando de um objeto em específico ou de um conjunto de objetos. Isso é feito a partir da especialização de GenericExportView nas classes GenericExportListView e GenericExportDetailView, respectivamente.

Figura 6 - Especializações de GenericExportView

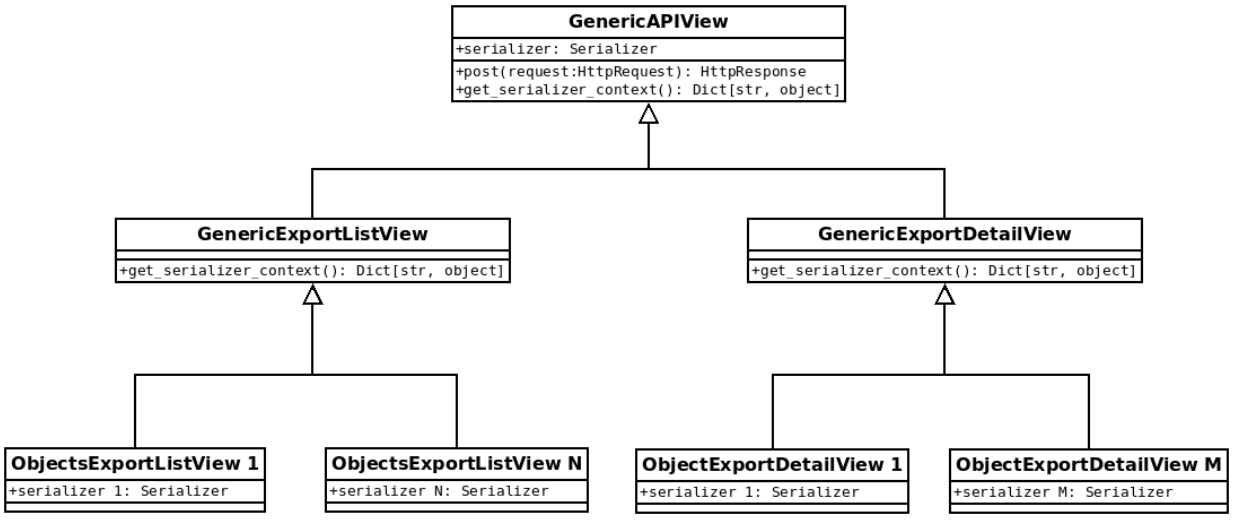
```python
class GenericExportListView(GenericExportView):
    def get_serializer_context(self):
        context = super().get_serializer_context()
        context["queryset"] = self.get_queryset()
        return context


class GenericExportDetailView(GenericExportView):
    def get_serializer_context(self):
        context = super().get_serializer_context()
        context["object"] = self.get_object()
        return context
```

Fonte: autoria própria

Todas as outras *views* são subclasses de uma das três classes anteriores. Do ponto de vista da *view*, o serializer deve apresentar apenas o método *export* e um atributo com o título do relatório, permitindo, dessa forma, encapsular a lógica de criação da planilha nas subclasses de serializer.

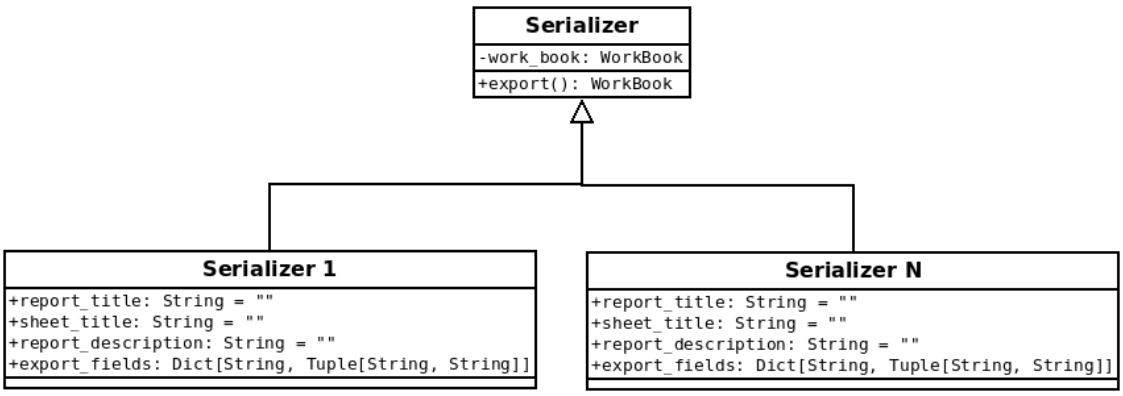
Figura 7 - Especializações de GenericExportView



Fonte: autoria própria

A classe base para os serializers define o método *export* genérico para a geração de planilhas que representam uma tabela no banco de dados. Sendo necessário apenas que subclasses de Serializer definam o atributo *export_fields*, que é um dicionário onde cada parâmetro recebido do módulo de gestão faz referência a uma coluna em determinada tabela no banco de dados.

Figura 8 - Especializações de Serializer



Fonte: autoria própria

Figura 9 - Classe base para todos os serializers

```
12  class Serializer(serializers.Serializer):
13      def __init__(self, *args, **kwargs):
14          super().__init__(*args, **kwargs)
15
16          self.work_book = xlwt.Workbook(encoding="utf-8")
17
18          for field in self.export_fields:
19              self.fields[field] = serializers.BooleanField(required=False, default=True)
```

Fonte: autoria próprio

Figura 10 - Método export da classe Serializer (Criação do Header)

```
21      def export(self, **kwargs):
22
23          user = self.context["request"].user
24          if not user.is_authenticated:
25              raise serializers.ValidationError('Context necessita de uma instância de
26
27          sheet = self.work_book.add_sheet(self.sheet_title)
28          queryset = self.context["queryset"]
29          style = xlwt.easyxf("font: bold on;")
30
31          # seta header
32          column = 0
33          for field in self.validated_data:
34              if field in self.export_fields and self.validated_data[field]:
35                  title, _ = self.export_fields[field]
36                  sheet.write(6, column, title, style)
37                  column += 1
38
39          # GOVERNO DO ESTADO XX
40          colspan = column if (column > 4 and column < 15) else 14
41          sheet.write_merge(
42              1, 1, 0, colspan, utils.sirafne_states[user.sirafne_uf],
43              xlwt.easyxf("font: bold on; align: wrap on, vert centre, horiz center"),
44          )
45
46          # NOME DA SECRETARIA
47          sheet.write_merge(
48              2, 2, 0, colspan, user.state.setting.secretariat_name,
49              xlwt.easyxf("font: bold on; align: wrap on, vert centre, horiz center"),
50          )
51
52          # DESCRIÇÃO
53          sheet.write_merge(
54              4, 5, 0, colspan, self.report_description,
55              xlwt.easyxf("font: bold on; align: wrap on, vert centre"),
```

Fonte: autoria própria

Figura 11 - Método export da classe Serializer (Preenchendo a guia da planilha)

```
56          )
57
58          row = 7
59          for item in queryset:
60              column = 0
61              for field in self.validated_data:
62                  if field in self.export_fields and self.validated_data[field]:
63                      _, field_name = self.export_fields[field]
64
65                      if hasattr(item, f"{field_name}_format"):
66                          value = item.__getattribute__(f"{field_name}_format")()
67                      else:
68                          value = item.__getattribute__(field_name)
69
70                      sheet.write(
71                          row, column, value.upper() if isinstance(value, str) else value
72                      )
73                      column += 1
74              row += 1
75
76          return self.work_book
```

Fonte: autoria própria

Figura 12 - Especialização da classe serializer

```python
class ExportIndividualProducerSerializer(Serializer):

    report_title = "Ofertantes Individuais"
    sheet_title = report_title
    report_description = "LISTA DE OFERTANTES INDIVIDUAIS"

    export_fields = {
        "name": ("Ofertante", "name"),
        "cpf": ("CPF", "cpf"),
        "gender": ("Gênero", "gender"),
        "date_of_birth": ("Data de Nascimento", "date_of_birth"),
        "dap": ("DAP Física", "physical_dap"),
        "self_declaration": ("Auto declaração", "self_declaration"),
        "funding_fida": ("FIDA", "funding_fida"),
        "funding_bid": ("BID", "funding_bid"),
        "funding_bird": ("BIRD", "funding_bird"),
        "state": ("UF", "state"),
        "city": ("Município", "city"),
        "territoriality": ("Territorialidade", "territoriality"),
        "street": ("Rua", "street"),
        "zipcode": ("CEP", "zipcode"),
        "district": ("Bairro", "district"),
        "number": ("Número", "number"),
        "complement": ("Complemento", "complement"),
        "reference_point": ("Ponto de referência", "reference_point"),
        "mobile": ("Celular", "mobile"),
        "phone": ("Telefone", "phone"),
        "mail": ("E-Mail", "mail"),
        "local_market": ("Feiras locais", "local_market"),
    }
```

Fonte: autoria própria

Relatórios que exigem um conjunto de regras mais complexas devem reescrever o método export de acordo com as particularidades de cada caso.

Figura 13 - Sobrescrita do método export

```
217  class ExportGroupSerializer(Serializer):
218
219      export_fields = {}
220
221      def export(self, **kwargs):
222
223          user = self.context["request"].user
224          if not user.is_authenticated:
225              raise serializers.ValidationError('Context necessita de uma instância de User
226
227          obj = self.context["object"]
228
229          description = self.report_description
230
231          products, _ = self.get_sheet_for(work_book=self.work_book, group=obj, initial=7)
232
233          export.create_sheet_header(
234              products,
235              sirafne_state=utils.sirafne_states[user.sirafne_uf],
236              description=description,
237              secretary=user.state.setting.secretariat_name,
238          )
239
240          return self.work_book
241
242      def get_sheet_for(self):
243          raise NotImplementedError
244
245      @property
246      def report_title(self):
247          return self.get_report_title()
```

Fonte: autoria própria

Caso a chamada ao método *export* seja bem sucedida, o objeto que representa a planilha é criado. Por fim, ele é convertido em uma cadeia de *bytes* e escrita na resposta, que então, é enviada ao módulo gestor.

6. CONCLUSÃO DO PROJETO

Ao final do projeto, em setembro de 2021, todos os itens definidos no cronograma inicial do projeto de software foram atendidos.

- Sistema de armazenamento
- Modelagem e Implementação de sistema de Banco de Dados
- Disponibilização de Manuais
- Publicação de API Pública
- Sistema de Mineração de Dados
- Publicação de API Privada

O SIRAF-NE está em funcionamento desde fevereiro de 2021, momento no qual foram importados os primeiros cadastros de agricultores provenientes do Estado do Rio Grande do Norte. Na data em que este relatório foi finalizado, 21 de março de 2023, o SIRAF-NE conta com o cadastro de 26536 agricultores, 66 grupos informais, 158 cooperativas, 283 associações e 14425 ofertas entre os 781 produtos disponíveis.

7. REFERÊNCIAS

AGRICULTURA FAMILIAR NO NORDESTE: UM BREVE PANORAMA DOS SEUS ATIVOS PRODUTIVOS E DA SUA IMPORTÂNCIA REGIONAL. Disponível em: https://repositorio.ipea.gov.br/bitstream/11058/10481/1/brua_23_artigo7.pdf. Acesso em: 19 mar. 2023.

DJANGO: Documentation. Disponível em: https://docs.djangoproject.com/. Acesso em: 19 abr. 2023.

Django Rest Framework: API Guides. Disponível em: https://www.django-rest-framework.org/. Acesso em: 19 abr. 2023.

XLWT: Documentation. Disponível em: https://xlwt.readthedocs.io. Acesso em: 19 abr. 2023.