**DataViva**

Guia de Desenvolvimento

V0.1

Núcleo de Sistemas e Gestão

Fevereiro 2013

Histórico de Revisão

|  |  |  |  |
| --- | --- | --- | --- |
| DATA | VERSÃO | DESCRIÇÃO | AUTOR |
| 29/07/2013 | 1.0 | Criação do documento | Mario H. Mol |
| 31/10/2013 | 1.1 | Documento base | Mario H. Mol |

ÍNDICE

[1. INTRODUÇÃO 4](#_Toc381117061)

[2. ARQUITETURA 5](#_Toc381117062)

[2.1. Dependências 5](#_Toc381117063)

[2.2. Componente de Gráficos 6](#_Toc381117064)

[2.3. Banco de Dados 8](#_Toc381117065)

[2.4. Tecnologias de Client-Side 8](#_Toc381117066)

[3. APLICAÇÃO 10](#_Toc381117067)

[3.1. Estrutura 10](#_Toc381117068)

[3.2. Inicializador 14](#_Toc381117069)

[3.3. Linguagem de Programação 14](#_Toc381117070)

[4. FORMULÁRIOS 16](#_Toc381117071)

[4.1. Visão 17](#_Toc381117072)

[5. CONTROLE 18](#_Toc381117073)

[5.1.1. Rotas 18](#_Toc381117074)

[5.1.2. Dados 19](#_Toc381117075)

[5.1.3. Navegação 20](#_Toc381117076)

[5.1.4. Template 20](#_Toc381117077)

[5.1.4.1. Blocos 21](#_Toc381117078)

[6. Model 22](#_Toc381117079)

[6.1. Banco de Dados 23](#_Toc381117080)

[7. Decorators 25](#_Toc381117081)

[8. VISÃO 26](#_Toc381117082)

[8.1. Dados 26](#_Toc381117083)

[8.2. Configurações Comuns 28](#_Toc381117084)

[8.2.1. Configurações de Agrupamento 29](#_Toc381117085)

[8.2.2. Configurações de Layout 29](#_Toc381117086)

[8.3. TreeMap 30](#_Toc381117087)

[8.4. Network e Rings 31](#_Toc381117088)

[8.5. Bubbles 32](#_Toc381117089)

[8.6. Geo Map 33](#_Toc381117090)

[9. INTERNACIONALIZAÇÃO 34](#_Toc381117091)

INTRODUÇÃO

Está sendo implantado em Minas Gerais o *Project Space* (PS), em parceria com a *Growth Ventures* (GV) e o Governo do Estado de Minas Gerais. Criado pelos professores Ricardo Hausmann (Universidade de Harvard) e Cesar Hidalgo (Instituto de Tecnologia de Massachusetts), no ano de 2007, o Product Space é um conjunto de ferramentas, conceitos e metodologia que tenta explicar o desenvolvimento desigual de estados ou regiões e criar instrumentos para que esse processo seja feito de maneira mais uniforme.

Uma entrega deste projeto é uma plataforma Web que disponibilizará para seus usuários uma ferramenta de pesquisa e análise de um grande volume de dados, utilizando a vanguarda da exibição de dados com gráficos de última geração. Os desenvolvedores da GV entregaram ao Escritório de Prioridades Estratégicas de Minas Gerais (EPE-MG) os fontes deste sistema e conhecimento para que seja possível dar continuidade na sua manutenção e evolução. Utilizando os fontes foi realizada uma análise para garantir que, novos desenvolvedores possam trabalhar neste código da melhor forma possível.

Esta análise do sistema tem por objetivo principal orientar o desenvolvedor que for trabalhar na evolução deste projeto, resultando neste documento. Sendo assim temos aqui um documento a ser utilizado como guia para a manutenação ou desenvolvimento de novas funcionalidades na plataforma.

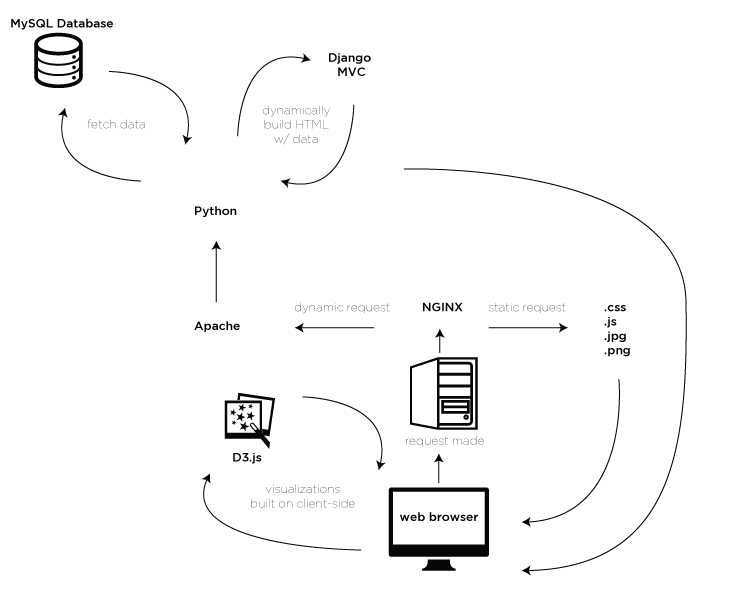
ARQUITETURA

Para este projeto foi utilizado a arquitetura *Flask*, escrita em *Python* e que proporciona grande produtividade no desenvolvimento e performance para uma grande número de acessos.

A versão utilizada neste projeto é o Django 1.4.3, que podemos encontrar mais informações em:

* https://www.djangoproject.com/

Veja um desenho básico de como funciona esta estrutura:



* 1. Dependências

Neste capítulo vamos tratar de todas as dependências a estes programas existentes para esta arquitetura.

Sistema operacional pode ser tanto Windows como baseados em Linux, como o Debian e Ubuntu. No caso a equipe de desenvolvimento da GV utiliza o Ubuntu Server, mas neste manual vamos tratar o sistema operacional Linux para montagem do ambiente de desenvolvimento, já que no Governo de Estado temos este padrão e pelo fato de no futuro este deverá ter competência para evolução do projeto.

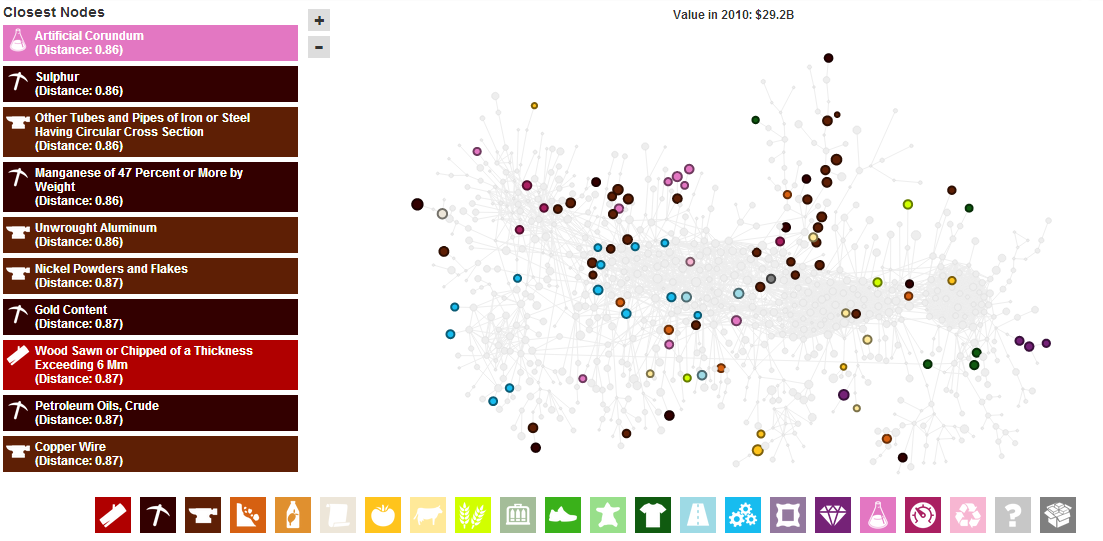
A linguagem em que o Django foi construído é o Python e a versão compatível é a Python 2.7.3 e que pode ser encontrado em:

* http://www.python.org /
  1. Componente de Gráficos

A camada de visão desta arquitetura é construída em duas partes. Primeiro é montado no Server-side por meio do Flask, a ser detalhado nos capítulos abaixo, e os relatórios/infográfos são construídos no client-side, através do D3 (<http://d3js.org/> ).

Os gráficos utilizados nesta solução são:

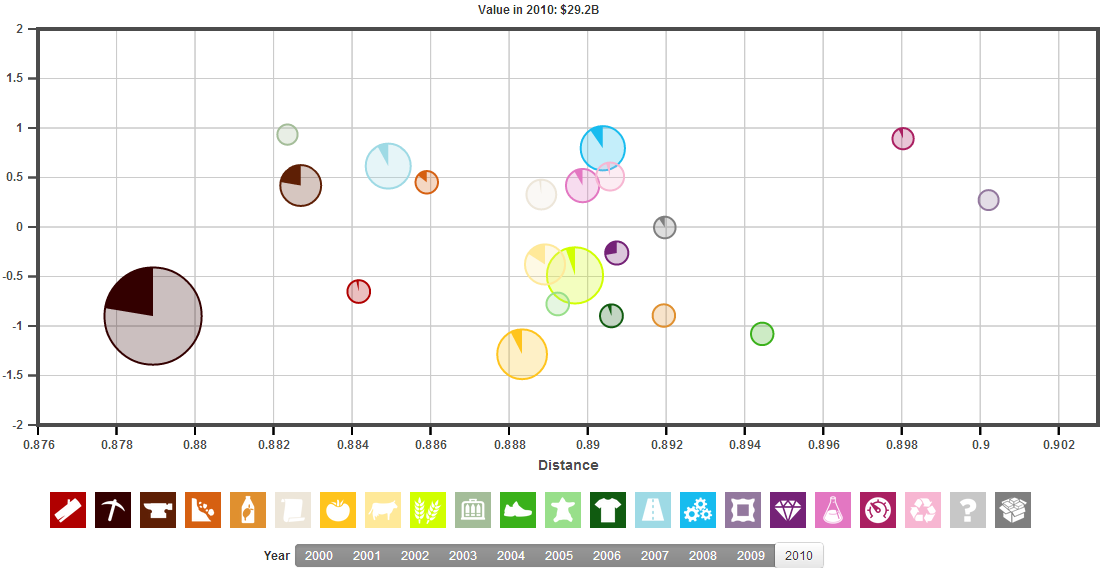
* Gráfico Network
  + Componente FishEye
  + <http://bost.ocks.org/mike/fisheye/>



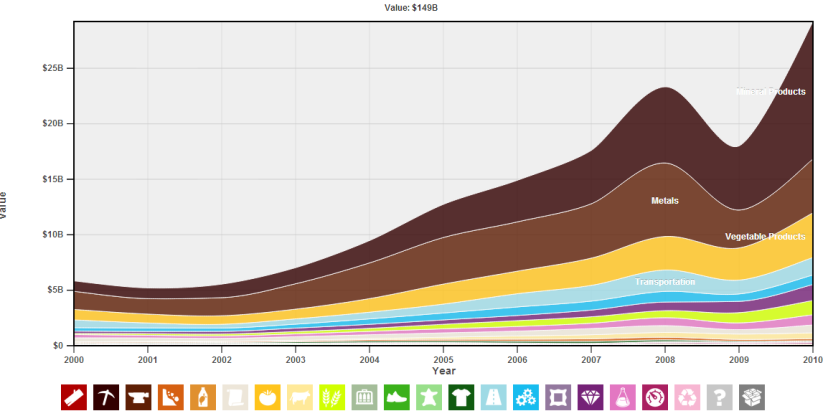
* Gráfico Treemap:
  + Componente Zoomable Treemap
  + http://mbostock.github.com/d3/talk/20111018/treemap.html



* Gráfico ScatterPlot:
  + Componente Scatter Plot
  + <http://bl.ocks.org/mbostock/3887118>



* Gráfico Stacked
  + Componente Stacked Area Chart
  + http://bl.ocks.org/mbostock/3885211



Existem outros componentes na Galeria, caso seja necessário utilizar novos conceitos nesta ferramenta.

* <https://github.com/mbostock/d3/wiki/Gallery>
  1. Banco de Dados

O banco de dados utilizado neste aplicação é o MySQL XXX e nele tem a estrutura de dados utilizado pela aplicação construída. Este framework em si precisa não precisa de tabelas de sistema, mas temos algumas como, por exemplo, para autenticação de usuário. Temos também tabelas com a massa de dados já calculados a partir das bases fornecidas pelo Estado.

Existe um driver próprio para conexão entre Python e o banco de dados MySQL. Este drievr pode ser encontrado para diversos sistemas operacionais e normalmente só a instalação deste driver é o suficiente para fazer funionar. Mas existem algumas particularidades a sobre uso do Mysql nesta arquitetura, caso precise de mais informações vide:

* + <https://docs.djangoproject.com/en/dev/ref/databases/#mysql-notes>

O tamanho esperado desta base de dados com todas as informações atualizadas é de 70 Gb. A base é relativamente grande, mas não a ponto de ser considerado um Big Data. O importante é que, como teremos apenas visualizações de informações o banco de dados pode ser otimizado para trazer resultados de forma muito rápida. A atualização das informações no banco de dados ocorrem duas vezes ao ano, portanto não tem problema que seja demorado para atualizar, pois em troca teremos consulta bem dinâmica.

* 1. Tecnologias de Client-Side

A tecnologia de cliente-side adotada neste projeto é o Javascript, portanto para todas as dependências apresentadas abaixo. A principal biblioteca e utilizada em todo o site é o D3, semelhante ao jQuery , ele cria atalho simples para um monte de funções javascript comumente utilizados. Onde D3 difere do jQuery é que ele foi feito com a visualização de dados em mente, portanto permite trabalhar com rapidez e eficiência com um grande volume de objetos. Para mais informações, vide a wiki do projeto:

* https://github.com/mbostock/d3/wiki

D3plus é uma extensão para a biblioteca D3 que permite a criação rápida e fácil das diferentes visualizações encontradas no DataViva . A documentação completa pode ser encontrada em:

* https://github.com/alexandersimoes/d3plus/wiki

Zoomable Tiles é um plug-in do D3 que permite atrabalhar com dados geográficos e o D3plus Geo Map depende dessa biblioteca.

* https://github.com/d3/d3-plugins/tree/master/geo/tile

TopoJSON é um outro plug-in D3 que permite a D3plus Geo Map usar arquivos topoJSON como coordenadas . TopoJSON ficheiros são mais rápidos , mais eficientes , e têm uma dimensão menor que os arquivos GeoJSON padrão. A documentação completa pode ser encontrada aqui . Para DataViva , os arquivos de forma ( . Shp ) para a geografia do Brasil foram obtidos no site do IBGE , convertido em GeoJSON , e depois convertido para topoJSON .

* https://github.com/mbostock/topojson/wiki

Queue.js é uma biblioteca auxiliar pequeno que permite que aos aplicativos DataViva invocar várias chamadas de dados assíncronos de uma vez, e esperar por eles todos para completar antes de prosseguir.

* https://github.com/mbostock/queue

Leon é uma biblioteca que converte os elementos de entrada HTML padrão (como botões de rádio, listas de seleção, e sliders alcance) em mais atraentes e cross-browser entradas uniformes. A documentação completa pode ser encontrada em:

* https://github.com/davelandry/leon/wiki

JFont Checker é uma pequena biblioteca que permite DataViva detectar quais fontes browser / suportes computador do usuário . Porque gráficos SVG não pode usar uma lista de fontes para verificar quando a renderização (como faz CSS), precisamos encontrar a melhor fonte compatível para uso nas visualizações.

* http://derek1906.site50.net/works/jfont.php

Redactor é utilizado nas partes de rich-text do sistema, como na página "Fale Conosco". Com ele é possível formatar os textos com HTML para palavras em negrito e itálico . Redactor requer jQuery mas é o único lugar em todo o sistema que será necessário o uso do jQuery. A documentação completa para Redactor pode ser encontrada em:

* http://imperavi.com/redactor/docs/

Para detectar quais recursos estão disponíveis no navegador do usuário, usamos Modernizr. DataViva utiliza Modernizr para detectar se o usuário está utilizando um dispositivo baseado em toque ou mouse, bem como informações sobre o suporte a browser a determinados recursos. Um exemplo de uso é verificar se o browser aceita atualização de url dinamicamente. A documentação completa pode ser encontrada em:

* http://modernizr.com/docs/

Moment.js é uma biblioteca que permite a fácil manipulação de data e hora. DataViva usa moment.js no "Fale Conosco" exibir datas como " 4 meses atrás " , ao invés de apenas mostrar a data da postagem . A documentação completa pode ser encontrada em:

* http://momentjs.com/docs/

APLICAÇÃO

A plataforma utilizada para a construção do sistema é o Flask, um framework criado em Python com o intuito de ser leve e de fácil uso. No início foi utilizando o framework Django, também em Python, mas para deixar a aplicação ainda mais leve foi adotado o Flask.

* 1. Estrutura

A pasta visual, dentro da pasta com todos os fontes, tem todo o código que compõe a plataforma Web. Dentro desta pasta existem várias pastas, onde cada uma funciona como um pacote de funcionalidades em comuns ou como arquivos estáticos que são utilizados por toda aplicação.

Uma coisa que é comum a estas pastas de pacote é que todas devem ter um Inicializador, Model, View e Form.

Estes arquivos são criados com o seguinte padrão:

* Inicializador:
  + \_\_init\_\_.py
* Form:
  + forms.py
* Model:
  + models.py
* View:
  + views.py
* Decorators:
  + decorators.py

As pastas que não existem como pacote de sistema, são utilizadas para gravar arquivos estáticos utilizados pela plataforma Web, como por exemplo os arquivos de tradução do sistema.

Nos próximos capítulos vamos detalhar mais informações sobre os objetivos e necessidade de cada arquivo deste, além de demonstrar como é possível realizar manutenções e novos desenvolvimentos nestes arquivos de sistema.

Na raiz desta pasta, visual, temos um arquivo inicializador (\_\_init\_\_.py) e um arquivo de funções úteis (utils.py), utilizado por todas as camadas desta aplicação.

Abaixo vamos listar todas as pastas, identificando se é de sistema e um breve descritivo sobre seu proposito no sistema.

**About**

Aqui ficam as telas de ajuda do sistema, bem como aquelas descrevendo cada aplicativo. Mesmo com maior parte deste conteúdo ser de textos e imagens, este ainda é um pacote de sistem.

**Account**

Nesta pasta ficam os códigos que realizam a autenticação do usuário na plataforma. Além dos arquivos padrão de um pacote de sistema, temos um arquivo chamado login\_providers.py , com código específico para autenticação em redes sociais.

**Admin**

Códigos que fazem a parte de administração do site, a ser acessado apenas por usuários restritos do governo. Aqui temos telas para administrar usuários e também para acompanhar e responder ver questões levantadas pelos usuários no site.

**Apps**

Temos aqui boa parte das navegações que são realizadas pelos usuários, como a criação dos Apps e do App Builder. Em tese, o App Builder faz chamadas sempre ao Embed. O embed por sua vez concentra a criação de todos os apps disponíveis em um só arquivo.

**Ask**

Neste pacote estão os scripts que fazem o Ask ao Avatar do sistema, isto engloba a visualização das perguntas e respostas , mas tem o mecanismo para realizar perguntas pelos usuários.

**Attrs**

Este pacote de sistema faz a leitura de atributos realizado pelo sistema. Estes atributos são identificadores para as localidades, as atividades econômicas, as classificações de ocupações entre outros.

**Data**

Neste pacote temos a parte de exportação de dados, tela disponível ao usuário para realizar filtros e exportação das informações do DataViva.

**General**

Esta pasta contém funcionalidades gerais e genéricas utilizado por todo o sistema. Um exemplo de funcionalidade encontrada aqui é a mudança de linguagem.

**Guide**

No sistema temos um guia que ajuda o usuário a montar a consulta, considerando passos de perguntas a serem respondidas. O código fonte que cria este guia pode ser encontrado nesta pasta.

**Html**

Esta pasta não é um pacote de sistema e contém arquivos de HTML utilizado para montar a visão do sistema. No caso todas as demais pastas de pacote de sistema utilizam os HTML’s desta pasta para criar a interface para o usuário, de acordo com a funcionalidade.

Para exemplificar, temos um diretório dentro deste chamado Data, com as interfaces que montam a listagem de dados bem como um menu para que o usuário possa realizar filtros dentro deste.

**Templates**

Esta pasta em específico que merece uma maior atenção, pois nela temos os HTML’s que geram o layout do sistema em sí. Ou seja, nas outras pastas de HTML temos códigos que geram uma tela em específico, como por exemplo o formulário de login. Mas nesta pasta template temos o código HTML que gera a tela em volta, ou seja, topo, menus e rodapés.

**Profile**

Perfil é uma parte do sistema que possibilita a busca de informações utilizando grupos de informações, como a localidade e as atividades econômicas. Portanto neste pacote de sistema temos o código que realiza a construção desta funcionalidade.

**Rais**

A construção dos gráficos e telas se dá principalmente pela pasta App, mas esta pasta realiza a recuperação de informações do banco por meio de outras pastas. Um exemplo de pacote que realiza a recuperação de informações é esta pasta Rais, onde temos consultas que retornam as informações da rais conforme foi requisitado pelo usuário.

**Secex**

Esta pasta tem a mesma formato de funcionamento que a Rais, mas retornando informações da Secex.

**Static**

Neste local temos arquivos estáticos, utilizados principalmente para criar interfaces em HTML, como imagens e cores. Temos as seguintes pastas e conteúdos:

* *CSS*: Aqui estão arquivos de estilo do site, chamados de CSS, onde são definidos cores, layout e fontes utilizadas.
* *IMG*: Imagens em diversos formatos, png, jpg e outros.
* *JS*: Arquivos javascript, utilizados na camada de visão e executado pelo navegador do usuário. Existe aqui frameworks já conhecidos e também códigos criados específico para esta solução.
* *JSON*: Aqui temos informações em formato JSON, que é um padrão de como gravar informações. No caso desta pasta temos arquivos com conteúdo de limites dos municípios ( coordenadas latitudinais) e também com a disposição dos objetos nos gráficos de network, no caso o Product e o Industry Space.

**Translations**

O site pode ser exibido em duas linguagens, inglês e português. Para que a plataforma possa funcionar independente da linguagem, existe esta pasta, onde temos arquivos textos nas diferentes linguagens a ser utilizado no site.

No caso é utilizado um padrão muito utilizado na Web e ambiente Linux, onde um arquivo de tradução com extensão .po é criado e outro compilado, com extensão .mo, também é criado.

Assim no /visual/translations/pt/LC\_MESSAGES/messages.po temos as traduções para o português (veja a pasta /pt/ ) e com conteúdo da seguinte forma:

#: visual/html/home.html:23 visual/html/templates/site.html:14

msgid "Guide"

msgstr "Guia"

Neste exemplo, temos na primeira linha um comentário, iniciado por #, e nas seguintes é dado um identificador (msgid) e seu respectivo valor na linguagem em questão (msgstr).

Para completar temos a pasta scripts, com conteúdo que dá suporte a aplicação como um todo. A estrutura encontrada nesta pasta é:

* Add\_to\_db
  + Nesta parte temos a parte do código que grava os valores encontrados no cálculo para o banco de dados
  + Temos as pastas attrs, rais, Secex, contendo scripts de importação de cada informação separadamente
* Calculations
  + Faz de uso da base de dados com os dados de entrada e aplica os cálculos dos indicadores.
  + Aqui temos uma pasta para cada valor de cálculo utilizado, contendo então as pastas complexity, distance, growth, importance, opp\_gain, proximity, rca, required, unique
* Geography
  + Geojson
    - Arquivos json com as delimitações geográficas de todos os estados
  + id\_conversion
  + merge\_geojson
    - Este script tem por objetivo unificar diversos json de geoposicionamento em um arquivo único
  + shape\_files
    - Outras formas a serem apresentadas em um mapa, para exemplificar temos a geolocalização de países, portos, estradas entre outros.
  + Topojson
    - Arquivos json com as delimitações geográficas de todos os municípios brasileiros, agrupados por estado
* Graphics
  + Script que faz a associação de imagens a aplicação Web. Hoje temos uma funcionalidade para tratar as bandeiras dos estados e países.
* Growth\_lib
  + Biblioteca com o processo de cálculo dos indicadores utilizados, como a taxa de crescimento, ganho de oportunidade, proximidade e o RCA.
* Utilities
  + Pasta contendo funções úteis a ser utilizado por qualquer parte do processo de cálculo, hoje temos apenas uma função que converte o arquivo de traduções em CSV para JSON, a ser utilizado na aplicação Web.
  1. Inicializador

Cada pasta considerada de pacote de sistema, no Python, deve ter o arquivo \_\_init\_\_.py . Portanto para que o compilador Python possa identificar que uma tal pasta é de sistema, esta precisa ter este arquivo.

Os arquivos inicializadores seguem então o seguinte padrão de localização no sistema:

* /visual/[nome\_do\_pacote]/\_\_init\_\_.py
  + Ex.: /visual/ask/\_\_init\_\_.py

Em todas as pastas de sistema este é um arquivo vazio, ou seja, ele existe apenas para identificar que a pasta deve ser lida pelo compilador do Python.

Não existe necessidade, nesta arquitetura, para a codificação de script de inicialização, portanto podemos tomar como padrão a criação deste como vazio.

* 1. Linguagem de Programação

A principal linguagem de programação utilizada nesta arquitetura é o *Python*, presente em todas as camadas da aplicação. Temos o uso do *Javascript* para a criação de gráficos interativos com o usuário, utilizando o processamento do navegador. Nesta parte do documento vamos tratar da linguagem principal da arquitetura, o Python. No capítulo sobre a camada de visão vamos detalhar melhor o uso do Javascript.

O Python tem por característica principal a redução do código, comprimindo funções e retirando necessidades de marcação de fim de estrutura.

Veja alguns exemplos abaixo:

* Sempre que utilizar um fim de linha (*enter*), ele já considera outra linha de comando, sem necessidade de usar ; por exemplo.
* A indentação conta, portanto ao usar o *tab* o desenvolvedor informa que aquela linha de instrução é filha da outra, em caso de um bloco IF por exemplo.

Além disso existe mudança em praticamente todas as instruções básicas, com o tratamento de *loop* e *array*.

* IF
  + O uso do if não tem uso de parêntesis , *(* *,* e utiliza no fim o dois pontos , *:*
    - Ex.: if "google\_id" in user\_fields:
  + Sem declaração de fim de IF e else if com elif

*if "google\_id" in user\_fields:*

*k="google\_id"*

*elif "twitter\_id" in user\_fields:*

*k="twitter \_id"*

*else*

*k="twitter \_id"*

* Loop em Array com Condicional:

*user\_fields = {k:v for k,v in user\_fields.items() if v is not None}*

* + Faz um loop com a lista presente em items(), lista esta de array, no caso Key and value , por isso o k,v.
  + A condicional está no final, onde o valor não é nulo.
  + user\_fields passa a ser uma lista de Key e Value com Value sempre preenchido
* Função

*def questions(nickname, status=None):*

*return nickname*

FORMULÁRIOS

Este é responsável por ter uma interface de comunicação entre o formulário em *HTML* do cliente e o framework Web. Portanto para cada *form* existente deverá existir esta classe com os seus respectivos campos.

Portanto para encontrar o arquivo do formulário é necessário acessar a pasta:

* /visual/[nome\_do\_pacote]/forms.py
  + Ex.: /visual/ask/forms.py

Como boa parte do sistema é feito por meio de visualização, não existem muitos *forms* no código. Em um arquivo *form* pode existir diversas classes de formulário utilizado no sistema, bem como seus respectivos campos e validadores. Mas as classes de formulário devem estar relacionadas ao pacote em questão, vamos exemplificar abaixo para facilitar o entendimento.

No caso do pacote Ask, onde temos funcionalidades de pergunta e resposta no sistema, existem um arquivo forms.py onde temos 3 classes de formulários:

* *SearchForm*: Formulário utilizado pelo usuário para realizar pesquisa de conteúdo no site
* *AskForm*: Formulário onde o usuário poderá inserir sua pergunta e enviar a plataforma web.
* *ReplyForm*: Neste formulário o usuário poderá responder a uma pergunta ou comentário no sistema.

Nos três exemplos acima temos formulários no mesmo arquivo, pois todos fazem parte do contexto *Ask*, por isso estão no mesmo pacote de sistema. Todos os formulários utilizam o mesmo componente do flask, chamado the *WTForms* e que provê todas as funcionalidades necessárias para uma rápida construção.

Veja que por padrão, a classe de formulário deverá terminar com a palavra *Form* como, por exemplo, *SearchForm*.

from flask.ext.wtf import Form, TextField, TextAreaField, HiddenField

from flask.ext.wtf import Required, Length

class SearchForm(Form):

search = TextField('search', validators = [Required()])

class AskForm(Form):

question = TextField('question', validators = [Required()])

body = TextAreaField('body', validators = [])

app = TextField('app', validators = [])

tags = TextField('tags', validators = [])

class ReplyForm(Form):

reply = TextAreaField('reply', validators = [Required()])

parent = HiddenField('parent')

No caso é o framework *Flask* que faz a captura dos dados vindos do usuário, através de um Post feito pelo navegador do usuário. Estes dados capturados são utilizados então para preencher uma classe form, como no exemplo acima. Por último o framework pega esta classe form preenchida, passa para a função da view que foi chamada pelo usuário, para então continuar o fluxo de processamento.

Para cada campo do formulário é possível determinar sua exibição e comportamento:

* Forma de exibição em tela: Escolhendo o componente a ser inserido no HTML da tela
  + Textfield(), TextAreaField(), HiddenField()
    - question = TextField('question', validators = [Required()])
* Primeiro argumento desta função é o identificador do campo
* O segundo é a lista de validações
  + Obrigatório:
    - validators = [Required()]
  + Limites: Tamanho máximo e mínimo
    - validators = [Length(min=0, max=256)]
  + Padrões: Deve seguir o padrão de uma URL
    - validators=[url()]
  1. Visão

A construção do form na página, portanto em código *HTML*, é possível ser feito utilizando o construtor de formulário do componente *WTForms* ou utilizando diretamente *HTML*. O componente possibilita também a sua utilização parcial, ou seja, construir o formulário em *HTML* e fazer a leitura na *views.py* por meio da classe de formulário no *forms.py*.

CONTROLE

Esta parte do sistema é responsável por criar endereços a serem acessados pelos usuários, fazendo a passagem das informações vindas do usuário para a camada Model. Nesta parte são definidas interfaces de comunicação entre cliente e aplicação, utilizando o Form como um contrato de comunicação entre eles. Para encontrar o arquivo do formulário é necessário acessar a pasta:

* /visual/[nome\_do\_pacote]/forms.py
  + Ex.: /visual/ask/forms.py
    1. Rotas

Para definir qual método da visão deverá responder por um endereço de acesso, é necessário utilizar a decorador @map.route. Por exemplo, se quisermos acessar a parte de Ask do sistema, existe o endereço /ask/questions , e quem define este endereço é:

* Abre o arquivo de view da Ask
  + /visual/ask/views.py
* Veja o método index(), onde existe um mapeamento ‘/questions’ , como no exemplo abaixo. Quer dizer que, acessando o nome do pacote (no exemplo ask) e depois o route definido no decorator @mod.route , temos então o endereço a ser acessado.
  + Neste caso /ask/questions

@mod.route('/questions/', methods=['GET', 'POST'], defaults={'page': 1})

def index(page):

Um mesmo método da View pode ter várias rotas diferentes, inclusive passando variáveis no método, que podem ser utilizadas no método. O código abaixo foi retirado do /visual/guide/views.py :

@mod.route('/')

@mod.route('/<category>/')

@mod.route('/<category>/<category\_id>/')

@mod.route('/<category>/<category\_id>/<option>/')

@mod.route('/<category>/<category\_id>/<option>/<option\_id>/')

@mod.route('/<category>/<category\_id>/<option>/<option\_id>/<extra\_id>/')

def guide(category = None, category\_id = None, option = None, subject = None, option\_id = None, extra\_id = None):

Neste exemplo temos que, o método guide pode ser acessado por diferentes links, passando variáveis de acordo com o link, como abaixo:

* /guide/
  + Neste caso estamos caindo na primeira regra de roteamento e sem passar qualquer parâmetro, portanto todos os argumentos da função guide terão o valor default como definido, no exemplo acima será None.
* /guide/Animais/
  + Aqui já estamos preenchendo a variável category com o valor Animais

Caso seja necessário há a possibilidade de restringir o método de acesso a esta rota, seja por *GET*, *POST* ou *PUT*.

* @mod.route('/questions/', methods=['GET', 'POST'], defaults={'page': 1})
  + Neste exemplo os métodos permitidos são GET e PUT

Além disso, utilizando o *defaults*, a rota consegue preencher informações de argumentos do método da view.

@mod.route('/questions/', methods=['GET', 'POST'], defaults={'page': 1})

def index(page):

Neste caso o método index recebe de entrada a variável page, e quando for acessado pela rota /questions/ ela será preenchida com o valor 1.

Por fim é possível ainda fazer uma validação de tipo de dados, como por exemplo requisitar que tal parâmetro da URL seja inteiro, para entrar na condição do roteamento.

* @mod.route('/questions/<status>/<int:question\_id>/', methods=['GET', 'POST'])
  + 1. Dados

Nesta camada de visão temos dados que podem ser obtidos pelos seguintes meios:

* *Formulário*: Aqui são informações de formulários que aparecem na tela para o usuário preencher, informações estas que são enviadas a essa classe views.py como variáveis em um escopo de POST.
  + Você deve criar uma classe form e então utilizá-la na tela em HTML
  + Dentro do método da camda de visão, obter as informações utilizando o comando:
    - reply\_form = ReplyForm()
    - parente = request.form["parent"]
  + Neste exemplo estamos declarando que no request temos o form com as especificações do ReplyForm e que estamos pegando o valor da variável parent
* *Request*: Aqui são informações que estão no link de navegação, como por exemplo meusite/?order=newest , onde queremos obter o valor da variável id.
  + Para obter a variável order, faça:
    - order = request.args.get('order', 'votes')
    - Onde votes é o valor default de order, caso este não exista no link. (ex.: meusite/?id=1)
* *Session*: A sessão é um recurso comum a linguagens Web, pois possibilita gravar informações específicas para um usuário no servidor de aplicação.
  + Um uso muito comum deste recurso é gravar o nome do usuário na sessão, após conferir no banco de dados se sua senha.
  + Assim reduz tempo de acesso a esta informação, que passa a ser reutilizada enquanto o usuário estiver logado ao site.
    - session['has\_access'] = True
    - has\_access = session['has\_access']
    1. Navegação

A parte de navegação nos permite direcionar o fluxo do usuário em telas e ações. Temos a forma de redirecionar simples, utilizando uma URL:

* return redirect('/admin/questions', status=previous\_status)

No lugar de uma URL manual, temos a opção de utilizar a função url\_for, que de acordo com a view e o método ele busca a url , utilizando dos routes, e então redireciona:

* return redirect(url\_for('.admin\_questions', status=previous\_status))

Veja que nestes exemplos temos a passagem de parâmetros para a nova visão. Além deste podemos criar retornos com tipos específicos, como um json. No exemplo temos uma lista de usuários chamado de user, que invoca a função de serializar para que a função jsonify possa criar um json e fazer o retorno do request desta forma.

* return jsonify( {'user': user.serialize()} )
  + 1. Template

Nesta camada de visão temos dados que podem ser obtidos pelos seguintes meios:

* return render\_template("general/access.html", form=formLocal).

Veja que é possível passer informações da view para o html, através dos parâmetros após a url. Ou seja, é possível incluir qualquer atribuição, no exemplo acima que a variável formLocal da view poderá ser acessada no html por meio do nome form, como no exemplo:

* {{ form.id }}
  + - 1. Blocos

Para criar uma partial view, onde um html é inserido na parte de código de outro html, é possível ser feito por duas formas:

* A primeira é através dos blocks, onde no html pai de template define uma região e um nome pra ela, como no exemplo do site.html

{% block content %}

{% endblock %}

* + Depois em outro html é possível inserir o valor a ser preenchido nesse espaço, bastando fazer:

{% block content %}

Olá

{% endblock %}

* A segunda opção é por meio de include. Vamos dizer que temos uma listagem onde cada elemento deverá ser impresso de uma forma customizada. Assim cria-se um foreach e um include de um html, que fará a impressão de cada elemento, como no exemplo:

{% for item in lista %}

{% include "box/item.html" %}

{% endfor %}

1. Model

Existem muitas formas de organizar a estrutura do sistema na arquitetura, o que resultou na definição de alguns padrões de projeto. Um padrão bem básico e considerado minimalista, portanto com o mínimo de organização necessário para manutenação de um projeto, é chamado *Model View Controller* (MVC). Este padrão é definido por uma *View*, com a parte de exibição de informações ao usuário, o *Model*, com a parte de regras de negócio e banco de dados, e por último *Controller*, que faz a navegação e intermediação entre a *View* e o *Model*.

A arquitetura utilizada neste projeto tenta de certa forma seguir a separação do código nestes moldes, mas não está seguindo à risca. O grande problema neste projeto é que não existe a separação da regra de negócio na camada *Model*, ela fica junto com a View. Isto de certa forma é ruim para o sistema, isto porque:

* Reaproveitamento de Código: Há muito menos reaproveitamento de código de regra de negócio, isto porque uma consulta ao banco ou regra de negócio utilizada em uma view não pode ser reaproveitada em outra view.
* Manutenção: Uma alteração ou correção de bug, que está descentralizada em em visões aumentam o esforço da correção.

Existem outros pontos que são prejudicados por utilizar desta forma, mas como o sistema foi construído assim faremos a documentação da sua situação atual. Pensando no futuro, em qual seria a solução ideal para separar estas funções de regra de negócio, o ideal seria criar em cada pacote de sistema um arquivo chamado services.py , junto com forms.py, views.py e models.py. Nesta classe *Services* teríamos métodos que seriam utilizados nas *Views* e que utilizaria os *Models*, para ter o mapeamento de banco de dados.

Mas existem casos onde este recurso é utilizado nesta aplicação, sendo todos aqueles atribuídos aos dados brutos do sistema, como o *Secex* e *Rais*. Existe algo parecido também no pacote attrs. Nestes exemplos foram criadas funções genéricas na camada de visão para realizar este padrão de serviço, a ser reutilizado no código.

Sendo assim a camada *Model* deste projeto contempla apenas do mapeamento de banco de dados, com toda a regra de negócio dentro do views.py. Existem alguns casos em particular que foram criados regras de negócio na camada *Model*, portanto vamos abordar aqui exemplos de como foi criado o mapeamento do banco de dados e exemplos de métodos de regras de negócio que foram criados no *Model*.

* 1. Banco de Dados

Dentro dos diversos pacotes de sistema temos um arquivo *models.py*, com a estrutura do banco de dados. Cada tabela do banco de dados é definida por uma classe, e seus campos são definidos por atributos desta classe. Como exemplo vamos utilizar o arquivo */visual/ask/models.py* , que define as tabelas de banco de dados responsáveis por criar o mecanismo de *Ask*.

class Vote(db.Model):

\_\_tablename\_\_ = 'ask\_vote'

type = db.Column(db.SmallInteger, primary\_key = True, default=TYPE\_QUESTION)

type\_id = db.Column(db.Integer, primary\_key = True)

user\_id = db.Column(db.Integer, db.ForeignKey(User.id), primary\_key = True)

def \_\_repr\_\_(self):

return '<Vote %r:%r by:%r>' % (self.type, self.type\_id, self.user\_id)

Dentro deste arquivo *models.py* temos diversas classes, como a *Vote*, *Question*, *Reply* entre outras, ou seja, o mapeamento em classes de todas as tabelas que são necessárias para o funcionamento do *Ask* no site. Neste exemplo temos apenas a classe *Vote*, que é representada no banco de dados pela tabela *ask\_vote,* definida na variável *\_\_tablename\_\_*.

Veja que existe também a definição de 3 campos, Type, Type\_id e User\_id, abaixo alguns comentários sobre o que foi utilizado na criação dos campos:

* Em primeiro lugar veja que todos os campos são inicializados a partir do uso da função db.Column()
* Primeiro argumento desta função é o tipo de dados que este campo tem por representação no banco
  + Temos aqui exemplos de db.SmallInteger e db.Integer.
  + Podemos encontrar outros exemplos como db.DateTime, db.Text(), db.String(), db.Float()
  + Para todos estes tipos é possível incluir também o tamanho máximo de dados, por exemplo, criando o db.String(255) limitamos a máximo de 255 caracteres neste campo do tipo String.
* Como segundo argumento a definição de chave para este campo
  + Chave primária definida como primary\_key = True
  + Chave estrangeira definida por db.ForeignKey(User.id), onde passamos como argumento a classe a o atributo que este campo se relaciona
    - Ou seja, neste exemplo o campo user\_id da tabela ask\_vote é uma referência ao atributo id da classe User.
    - Veja que o nome real da tabela e do campo estão definidos dentro desta classe User
* Podemos ter o uso da chave primária e também da chave estrangeiro para o mesmo campo, o que acontece no exemplo do user\_id
* Temos ainda a definição de um valor default, caso ao grava o valor informado seja nulo
  + default = 1

Quando realizamos o mapeamento de um campo como chave estrangeira, devemos inserir também na classe relacionada. No exemplo do Ask, temos o campo user\_id que relaciona com User, sendo assim na classe User devemos criar um campo que representa o relacionamento com a Ask. Aqui como é feito o relacionamento na classe User:

* votes = db.relationship("Vote", backref = 'user', lazy = 'dynamic')
  + Veja que aqui é criado uma lista de votos, onde no primeiro argumento temos o nome da classe que realiza o relacionamento, neste caso a Vote
  + O argumento lazy determina se o componente de banco de dados do projeto deve ler estas informações automaticamente ou não
    - Por exemplo, ao buscar informações de um usuário, espera-se que já venha preenchido com a lista de votos dele, ou se deve deixar vazio e o desenvolvedor deve preencher manualmente.
    - Este tipo de configuração é muito importante para performance, ou seja, se o lazy for configurado para sempre preencher este campo, para uma simples consulta para saber o nome do usuário o sistema passa a sobrecarregar por baixar uma gama de informações que, neste momento é desnecessária.

Além da configuração básica de um relacionamento, temos a possibilidade de criar algumas outras particularidades no relacionamento de banco de dados, veja alguns exemplos abaixo:

* replies = db.relationship("Reply", backref = 'question', lazy = 'dynamic', order\_by="Reply.parent\_id")
* votes = db.relationship("Vote", primaryjoin= "and\_(Question.id==Vote.type\_id, Vote.type=={0})".format(TYPE\_QUESTION) , foreign\_keys=[Vote.type\_id], backref = 'question', lazy = 'dynamic')

Existem ainda outros atributos que o desenvolvedor pode utilizar para configurar seu mapeamento com o banco de dados, como por exemplo

* \_\_public\_\_ = ("id", "nickname", "email", "fullname", "role")
  + Neste exemplo definirmos quais campos, dentre todos os existentes no mapeamento, são públicos
* \_\_searchable\_\_ = ['question', 'body', 'status\_notes']
  + Aqui há uma declaração de restrição, informando quais campos são passíveis de serem utilizados em pesquisa no banco

1. Decorators

Decorator é um recurso do Python, que também é utilizado por outras linguagens, que permite você adicionar novos recursos a funcionalidades já existentes.

Existe apenas um uso de Decorator no sistema, no pacote Guide. Outros arquivos com este padrão foram criados mas eles estão vazios.

Portanto para encontrar o arquivo do formulário é necessário acessar a pasta:

* /visual/[nome\_do\_pacote]/decorators.py
  + Ex.: /visual/guide/decorators.py

O decorator criado nesta aplicação serve para conferir se o usuário ter permissão de acesso a visualizações do sistema. Sendo assim foi criado um decorator chamado *requires\_login* , como no exemplo abaixo:

from functools import wraps

from flask import g, flash, redirect, url\_for, request

def requires\_login(f):

@wraps(f)

def decorated\_function(\*args, \*\*kwargs):

if g.user is None:

flash(u'You need to be signed in for this page.')

return redirect(url\_for('users.login', next=request.path))

return f(\*args, \*\*kwargs)

return decorated\_function

Vamos dizer que a tela de Data deve ser então acesso apenas por usuários com login, para isso basta acrescentar no método de visão o novo decorator, como no exemplo:

* Abrir o arquivo view do pacote Data
  + /visual/data/views.py
* Encontrar o método de visão inicial, no caso aqui o Index
  + Adicionar este método o decorator criado
  + @requires\_login

@mod.route('/')

@requires\_login

def index():

return render\_template("data/index.html")

VISÃO

Grande parte da visão construída para o usuário é formado por componentes comuns a qualquer sistema Web, como CSS, HTML e Javascript. Mas temos neste projeto um diferencial na visualização das informações, que utiliza como premissa criar novas formas de visualizar dados utilizando oque há de mais novo em componente Web.

Sendo assim a equipe da GV pegou um componente chamado D3, especialista em inovação na visualização de dados, e criou novas funcionalidades, como se fosse uma evolução deste chamada D3 Plus.

* 1. Dados

Para a construção do gráfico são necessários duas massas de dados, uma com os dados de atributos e outra com informações de valores. É necessário que exista um campo com o mesmo nome nas duas bases, para então fazer o relacionamento entra elas na construção do gráfico.

Nos exemplos abaixo utilizarei as seguintes bases de dados:

* ISIC: Informações sobre as classificações de atividades econômicas, portanto é uma base de atributos
  + - Chamado por data/attr\_isic.json
    - Veja que cada elemento tem informações como nome, cor e o ID, no caso de um BI seria uma dimensão de informação

{ "data": [

{

"available": true,

"name": "Social Sciences R&D",

"color": "#e377c2",

"desc": "Research and experimental development on social sciences and humanities",

"num\_emp": 75404,

"keywords": "humanities",

"id": "m7220"

},

{

"available": true,

"name": "Other Wholesale",

"color": "#17becf",

"desc": "Non-specialized wholesale trade",

"num\_emp": 717552,

"keywords": "artist equipment, art, decorative, decoration",

"id": "g4690"

}

]

}

* ISIC MG: Informações sobre valores de complexidade, distâncias e RCA de um determinado ISIC para Minas Gerais, portanto aqui estão informações dos valores
  + - Chamado por data/isic\_mg.json
    - Veja que os valores estão em campos como complexity, rca e distance
    - Em BI estes dados seria o fato

{ "data": [

{

"wage": 535881.0,

"distance": 0.477044,

"num\_emp": 1122,

"complexity": 17.4386,

"isic\_id": "a0111",

"bra\_id": "mg",

"num\_est": 468,

"year": 2002,

"rca": 0.717473

},

{

"wage": 3203240.0,

"distance": 0.478277,

"num\_emp": 6578,

"complexity": 21.6126,

"isic\_id": "a0112",

"bra\_id": "mg",

"num\_est": 2804,

"year": 2002,

"rca": 1.87144

},

]

}

Nesta parte vamos mostrar como trabalhar na camada de visão considerando o framework Flask e utilizando o D3plus para todos os tipos de gráficos utilizados no sistema. Em todos os casos a construção do gráfico começa passando quais são as entradas de dados necessários, considerando que vamos padronizar o uso do Json como tipo de dados. Aqui temos a lista de atreibutos do ISIC gravado na variável attr e os valores do ISIC\_MG gravados na variável data.

d3.json("data/attr\_isic.json", function(attr){

d3.json("data/isic\_mg.json", function(data){

})

})

Se houverem duas listas com diferentes atributos, é necessário unifica-las em apenas uma. Desta forma, as variáveis com mesmo nome, mesmo vindo de fontes de dados distintas, são uma forma de relacionar os dados. Veja que criamos na lista de atributos (attr\_isic), um campo chamado isic\_id com o mesmo valor do id, só para relacionar com a base ISIC\_MG (isic\_mg) que tem o ID do ISIC com nome isic\_id.

var attrs = {}

attr.data.forEach(function(a){

a.isic\_id = a.id

attrs[a.id] = a

})

* 1. Configurações Comuns

Boa parte dos gráficos compartilham os mesmos atributos e configurações, portanto vamos falar aqui do que é comum a todos. Dai em diante em cada gráfico vamos detalhar o que é específico dele. Portanto chamamos a classe do viz, passando as seguintes informações:

*Type*: Tipo do gráfico, no caso o tree\_map

*Attrs*: Aqui deve ter uma lista com todos os atributos utilizados no gráfico, que será devidamente relacionada com a de dados por meio de campos com mesmo nome.

*Id\_Var*: Define o atributo que irá identificar a informação como única, ou seja, não será agregada e todos os isic\_id diferentes encontrados em dados será exibido no gráfico.

*Text\_var*: Além do identificador temos também o nome dado a cada entidade exibida.

*Value\_Var*: Com o identificador e o nome, agora selecionamos o valor que representa esta entidade. No exemplo abaixo foi escolhido o num\_est (número de estabelecimento), ou seja, será exibido os ISIC de Minas Gerais com o tamanho dado por meio da quantidade de estabelecimento.

*Tooltip\_info*: Aqui são passados atributos e valores de dados para serem exibidos em um tooltip que é exibido ao colocar o mouse em cima do objeto. No exemplo temos os atributos utilizados e agrupados por tipo de dados, portanto ao imprimir ele faz a conversão.

Click\_function: Quando um elemento no gráfico for clicado pelo usuário, existe a possibilidade de criar uma função javascript que realiza uma ação. Passando o nome da função como parâmetro desta chamada, conforme o exemplo abaixo:

* + - * .click\_function(clicker)

A definição desta função pode ser feita da seguinte forma:

clicker = function(obj) {

          console.log(obj)

          return "This is some test HTML"

        }

* + 1. Configurações de Agrupamento

Alguns gráficos permitem a criação de informações obtidas ao agregar informações.

.nesting(["nesting\_1","nesting\_3","nesting\_5"])

.nesting\_aggs({"distance":"mean","complexity":"mean"})

.depth("nesting\_5")

Neste exemplo temos 3 níveis de profundidade, chamados de 1, 3 e 5, veja na chamada da função nesting(). Os valores que permitem a soma das informações, são declaradas por meio do nesting\_aggs(), neste exemplo as variáveis distance e complexity podem ser agregados por meio da função média (mean). A declaração de qual profundidade deve ser utilizada como a inicial, é por meio da função depth().

Temos ainda a possibilidade de criar uma barra totalizadora, onde formata-se o valor e defini prefixo e sufixo:

.total\_bar({"prefix": "Export Value: $", "suffix": " USD", "format": ",f"})

Para formatação dos dados, existe uma função de number\_format e outra de text\_format, além de um estilo próprio a ser utilizado durante o carregamento do gráfico.

.number\_format(function(d){return "$"+Math.round(d)+" USD"})

.loader\_style({"color":"black","font-size":"10px"})

.text\_format(function(d){return d.toUpperCase()})

* + 1. Configurações de Layout

A visualização do gráfico pode ser customizado de diversas formas, seja na fonte utiliza ou no título exibido. As funções mais utilizadas são:

Title e Subtitle: Define um título do gráfico, exibido centralizado e no topo

* + - * .title("Product Space for Minas Gerais")
      * .sub\_title("Cool example of a sub-title")

Font e Weight: A fonte utiliza nos textos e também o seu estilo, se bold ou normal.

* + - * .font("Helvetica Neue")
      * .font\_weight("lighter")
  1. TreeMap

O treemap é um dos gráficos mais simples de ser criado e neste exemplo começamos pegando duas fontes de dados, os atributos e os dados.

d3.json("data/attr\_isic.json", function(attr){

d3.json("data/isic\_mg.json", function(data){

A chamada para criação do gráfico treemap contém apenas as informações primordiais, como o id\_var, attrs, text\_var e o value\_var. Outros gráficos simples utilizam esta mesma estrutura de dados.

viz

.type("tree\_map")

.id\_var("isic\_id")

.attrs(attrs)

.text\_var("name")

.value\_var("num\_est")

.tooltip\_info({"short": ["wage"],"long": ["distance", "complexity", "wage"]})

.name\_array(["name","id"])

.title("Product Exports of Minas Gerais")

.total\_bar({"prefix": "Export Value: $", "suffix": " USD"})

.year(2010)

.footer("Data provided by SECEX")

.color\_var("color")

d3.select("#viz")

.datum(data.data)

.call(viz)

* 1. Network e Rings

A construção do gráfico do tipo Network e Rings exige que a rede seja informada como uma entrada de dados. Esta rede deve estar completamente declarada, isto é, os nodes, suas conexões e a localização no espaço.

O arquivo network\_hs.json contém a rede, no attr\_hs temos os nomes e demais atributos dos produtos, e por fim o products\_mg contém os valores.

d3.json("data/attr\_hs.json", function(attr) {

d3.json("data/network\_hs.json", function(hs) {

d3.json("data/products\_mg.json", function(ps\_data) {

A chamada do D3 para criar o gráfico pede a passagem de dois parâmetros, o links e nodes, que nada mais é que a posição dos nodes e das suas conexões.

viz

.type("network")

.links(viz\_links)

.nodes(viz\_nodes)

.attrs(attr)

.value\_var("val\_usd")

.name\_array(["name"])

.tooltip\_info(["id","val\_usd","val\_quantity","val\_kg","complexity","distance","rca"])

.year(2010)

d3.select("#viz")

.datum(ps\_data)

.call(viz);

* 1. Bubbles

O gráfico e Bolhas (Bubble) é bem parecido com o Network, utiliza as mesmas configurações básicas. A sua única diferença é que ele possibilita que informações sejam agrupadas em uma bolha única. Para definir qual variável que cria este agrupamento, basta incluir o método grouping(), conforme o exemplo abaixo.

viz = d3plus.viz()

.type("bubbles")

.id\_var("id")

.attrs(attr)

.value\_var("val\_usd")

.text\_var("name")

.active\_var("employed")

.grouping("category")

.year(2010)

d3.select("#viz")

.datum(data2010)

.call(viz);

Neste caso, temos a categoria do produto e queremos que esta variável seja utilizada para realizar o agrupamento de produtos. Sendo assim, teremos grandes bolhas como categorias e dentro delas, pequenas bolhas contendo o valor de cada produto. No caso o tamanho da bolha é definida pela variável value\_var(), portanto neste exemplo o val\_usd. Assim quanto mais exportações houver de um determinado produto, maior será sua bolha.

* 1. Geo Map

Este é um dos gráficos de maior complexidade, afinal ele faz a plotagem de formas geométricas em cima de um mapa cartográfico, o que necessita de um mapa além dos objetos, comum a outros gráficos. Portanto as formas geométricas dos municípios são obtidos através do mg\_munic.json e passado pela função coords(). O nome dos municípios e demais informações de atributos, estão em attr\_munic e são passados ao D3 por meio do attrs().

d3.json("data/geo/mg\_munic.json", function(shapes) {

d3.json("data/munic\_a0112.json", function(geo\_data) {

d3.json("data/attr\_munic.json", function(attr) {

viz = d3plus.viz()

.type("geo\_map")

.id\_var("bra\_id")

.coords(shapes)

.attrs(attr)

.value\_var("wage")

.text\_var("name")

.tooltip\_info(tooltips)

.click\_function(clicker)

.year(2010)

d3.select("#viz")

.datum(geo\_data)

.call(viz);

* 1. Infinity Scrolling

Este é um dos gráficos de maior complexidade, afinal ele faz a plotagem de formas geométricas em cima de um mapa cartográfico

INTERNACIONALIZAÇÃO

Como esta plataforma é disponibilizada aos usuários em duas línguas, português e inglês, foi necessário instala um componente próprio para este fim. O Flask-Babel realiza todo este trabalho de forma bem objetiva, com ferramentas que auxiliam a criar todo o processo de tradução de um sistema. As configurações ficam em um arquivo como babel.cfg e o mínimo a ser informado é o padrão de extensão de arquivos python, de arquivos de visão (jinja2) e a extensão do Babel a utilizar:

* [python: \*\*.py]
* [jinja2: \*\*.html]
* extensions=jinja2.ext.autoescape,jinja2.ext.with\_

Na visualização, portanto ao construir o HTML, textos são inseridos junto com o código. Para que o sistema possa realizar a tradução do texto, é necessário escrever da seguinte forma:

* *<h1>{% trans %}Criar{% endtrans %}</h1>*

Veja que junto com o código HTML existem textos que devem ser traduzidos, neste caso o texto *Criar* foi colocando entre a tag *trans* e *endtrans*. Existe também a possibilidade de fazer a tradução dentro de um código python, por meio do uso da função gettext().

* return render\_template(“teste.html”, mensagem=gettext(“Criar”) )

Após criar todos os termos com esta tag, o Babel reconhece estes termos e cria um arquivo único com todas as traduções do sistema. Para extrair estas informações do projeto, é necessário executar o seguinte comando na pasta raiz do projeto:

* *pybabel -v extract -F babel.cfg -o messages.po ./*

Desta forma todos os arquivos na pasta projeto, são lidos e extraídos as tags, resultando em um arquivo messages.po com um conteúdo parecido com o abaixo:

* #: dataviva/ask/views.py:92 dataviva/ask/views.py:120
* msgid "You need to be logged in to vote."
* msgstr "Você precisa estar logado para votar"

Neste exemplo temos informando que o texto encontrado, informado com o msgid, foi encontrado nos pontos de código informado por meio do #. A tradução deste termo, para a linguagem buscada, é feita informado através do msgstr.

Após realizar toda a tradução, portanto preenchendo o msgstr, o arquivo final deve ficar na pasta relativa a sua linguagem, como:

* \translations\pt\LC\_MESSAGES

A criar de arquivo por linguagem pode ser feita pelo script:

* pybabel init -l en\_US -d ./translations -i ./messages.pot

Por fim este arquivo de mensagens deve ser compilado para o padrão de internacionalização, sendo um arquivo .mo. Para criar este arquivo é necessário executar o comando:

* pybabel compile -f -d ./translations

Um comando muito útil é a atualização dos arquivos de tradução. Vamos dizer que já foi realizada uma extração e tradução dos termos, mas depois de um tempo novos termos foram inseridos. Para que estes novos termos sejam adicionados no arquivo de tradução sem prejudicar o que já foi traduzido, é necessário realizar uma atualização:

* pybabel update -i messages.po -d pecbrasil/translations

A configuração da plataforma, além dos passos de tradução acima, é bem simples, começando pelo arquivo config.py

''' Available languages '''

LANGUAGES = {

'en': 'English',

'pt': 'Português'

}