Vanessa Development Update

- 2018.02.09.ET; last update 2018.03.26.ET

Introduction

The new Vanessa application requires a foundational 'toolkit' that meets the specific needs of ICAM. "v3" is the name I have given to the nascent toolkit that I am developing. This document describes v3 and highlights some architectural features and design aspects of v3.

V3 Javascript Toolkit

The current development/test url for V3 is:

http://icamlinux.surg.med.umich.edu/et_test_d3/suggest.html

V3 makes use of d3:

Fig. 0. The D3 Javascript library.

Quoting the D3 project web page,

"D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation."

D3 is a powerful and popular javascript library for data visualization. Edward (Brown) has also begun using D3 for data visualization of ICAM data. D3 shares certain paradigm features with other modern Javascript libraries like jQuery, but unlike jQuery, D3 is equally capable when manipulating HTML DOM elements as it is when dealing with SVG. Libraries like jQuery do not have the same facilities to handle SVG and in general also lack the data-driven paradigm that is one of D3's greatest strengths.

So the idea is to take advantage of D3 in our own development work.

In addition, v3 takes advantage of a number of new features that have been introduced into the Javascript language since *ECMAScript 2015* (*aka ES5*), including ES5's new *class* syntax, *arrow functions*, improved *variable scoping, destructuring assignments,* and *default parameters.*

As of this writing (2018.02.09), V3 is divided into several logical sections, objects, methods, and classes:

(1) v3.static: holds certain static objects that may be reused repeatedly by various instantiated objects. Currently, there are only two object references here:

1) `v3.static.tooltip`, and
2) `v3.static.ddlc`, which is the "dropdown list container" for the suggest list component.

(2) V3.KB: provides a list of common keyboard key code constants, e.g.: `v3.KB.enter, v3.KB.tab,` `v3.KB.up` (arrow key), `v3.KB.left` (arrow key), etc.

(3) v3.init: initializes v3 by creating certain reuseable SVG, HTML, and Javascript objects:

* SVG linear gradients
* SVG filters (such as a drop shadow)
* a reusable v3.Tooltip class object used by the v3.input group of classes
* a reusable v3.ddlc dropdown list container used by some of the v3.input group of classes

(4) v3.util: contains a set of utility functions that may be used by various v3 classes. For example:

```
v3.util.getViewportDimensions()
v3.util.setFocusToNextInput()
v3.util.makeFetchDataFactory()
```

(5) v3.svg: contains methods to add certain non-primitive SVG components, e.g:

`v3.svg.addCircledLetter_T_to()` which is used by `v3.input.Date`.
`v3.svg.addCircledDownArrowTo()` which is used by `v3.input.Suggest`.
... etc. ...

(6) **v3 classes**: Note that the "input" classes are grouped together into an "input" section. These classes create the user interface objects:

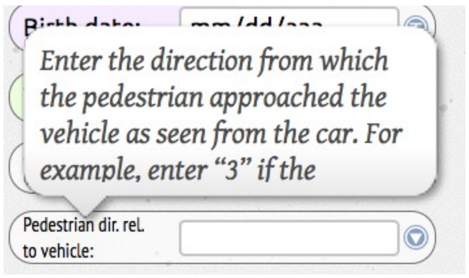
```
* v3.Tooltip
* v3.input.Basic
* v3.input.Number
* v3.input.Date
* v3.input.Suggest
```

User Interface (UI) element classes

The User Interface (UI) element classes are described below.

Tooltip

The `v3.Tooltip` class is used by other UI element classes to provide detailed descriptions or usage instructions on how to properly enter data. For example, the following tooltip describes how to enter data in the “Pedestrian direction relative to vehicle” field (*fig. 1*):



Enter the direction from which the pedestrian approached the vehicle as seen from the car. For example, enter “3” if the
Pedestrian dir. rel. to vehicle:

Fig. 1. Example of the v3.Tooltip class.

Note that one can scroll the text within the tooltip in cases where the text description is long — as is the case in the above example.

Input.Basic

The `v3.input.Basic` class provides a basic input element where one can enter unchecked textual or numeric data.

All of the other `v3.input` classes are sub-classes which inherit from `Basic`. Therefore, all of the UI input components share several useful features in common with `Basic` which are described below:

Automatic fluid repositioning. The input elements automatically “flow” to fill available space. Therefore, on wider screens or devices one might see 3 or even 4 input elements per line (*fig. 2*):

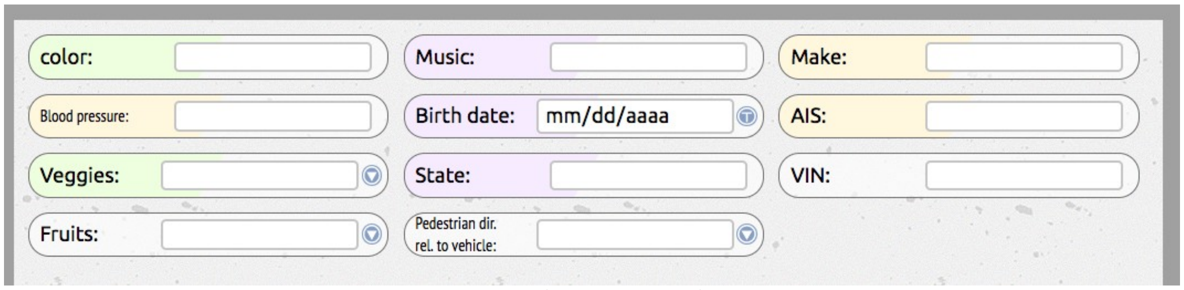


Fig. 2. Example UI on a wide screen.

... but on narrower screens or devices, only one or two input elements will occur per line (*fig. 3*):

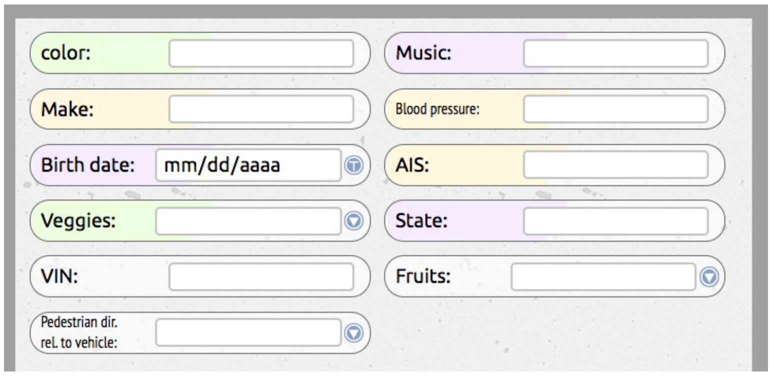


Fig. 3. Same UI example on a narrower screen or device.

The UI elements also scale themselves to a degree as well.

The advantage of all of this is that an application built using v3 can be used on any sized screen or device regardless of screen size or pixel density.

Color Tagging. The UI elements can also be "tagged" with different colors. As an example, in the sample UI shown previously (*fig. 3 above*), the green background (*"color" and "Veggies"*) can be used to represent VIPA-specific fields, while yellow (*"Make", "AIS", etc.*) might represent fields that are common to both VIPA and ICAM. Fields in lavender (*"Birth date" etc.*) can be used to highlight electronic patient health information (ePHI).

Tooltips. Clicking on the label or ground portion of a UI component toggles a tooltip which can be used to provide detailed notes on how to enter a particular field of information (*fig. 1*).

Labels. Often you will want to provide a string as the `label`. However, if you don't provide a `label`, then a default `label` will be constructed for you based on the provided `column` parameter. For example, if `column='pedestrian_impact_point'`, then `v3` will construct the label as "*Pedestrian impact point*" by capitalizing the first term and removing the underscores in the column name.

Labels Automatically Resize To Fit. Another feature of the `v3.input` UI component set is that if a label for an input element is long, the font is automatically changed to a narrower font and the type size is also reduced to insure that the label will fit. An example is the "*Pedestrian dir. rel. to vehicle*" label:

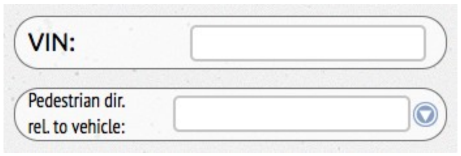


Fig. 4. Long labels are automatically resized to fit.

Input.Number

`v3.input.Number` is designed for recording continuous numerical data. In addition to specifying min and max values, one can also specify warnLow and warnHigh values. Entering a value in the range between min and warnLow or between warnHigh and max will result in the entry being flagged in yellow (fig. 5):

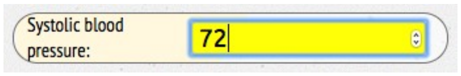


Fig. 5. Unusually low or high values which are still considered possible may be flagged in yellow as a warning indicator.

Entering a value less than min or greater than max will result in the entry being flagged in red (fig. 6):

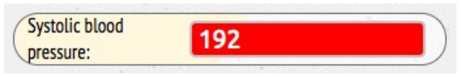


Fig. 6. Out-of-range low or high values may be flagged in red as an error indicator.

Input.Date

The `v3.input.Date` class allows entry of dates. The dates are entered according to the current locale. For example, the following screen shot is from a computer set for a Mexican Spanish locale (*fig. 7*). Regardless of the locale, the data will always be sent to (and received from) the database server uniformly in ISO *yyyy-mm-dd* format.

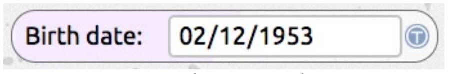


Fig. 7. The Date class showing Feb. 02, 1953 (Mexican Spanish locale).

As is also the case with the `v3.input.Number` class, one can specify *min, warnLow, warnHigh*, and *max* values for date entries too. Entries will be flagged in yellow or red accordingly.

In forms it is often necessary to specify today's date. Pressing the circled "T" button just to the right of the date entry input will populate the field with today's date (*fig. 8*):

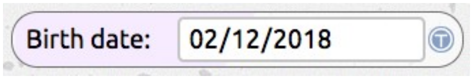


Fig. 8. Press the circled "T" button as a short cut to enter today's date.

INPUT.SUGGEST

The `v3.input.Suggest` class provides a dropdown suggest list as the user begins typing entries.

Match options. `Suggest` currently supports three `match` options:

`'b'`	— match from the **beginning of a string**
`'w'`	— match from the **beginning of words**
`'a'`	— match **anywhere in a string**

By default, the component will match from the beginning of a string (*option* `'b',` *fig. 9*):

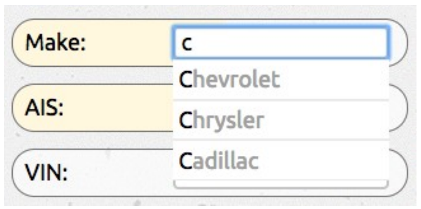


Fig. 9. The Suggest list matches from the beginning of a string by default.

A common use case for matching from the beginning of words, `'w'`, is for VIPA data using the GIDAS coding format in which entries begin with a numerical code and then are followed by a textual description. For example, for *gender*, users may prefer to type the letter "*m*" to match "*3 – **m**ale*".

Matching from any position is also supported via the `match='a'` option (*fig. 10*):

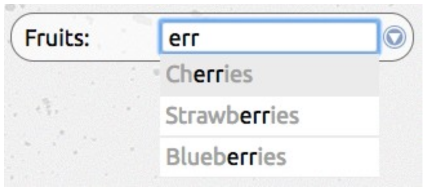


Fig. 10. A Suggest list with matchAnywhere set to true.

A good use case for matching anywhere occurs when creating a "smart" entry component for medical codes such as ICD-9, ICD-10, or AIS codes. For example, a user typing in a term like "*thorax*" will see matches to both "*thorax*" and "*pneumothorax*".

Sort options. `Suggest` currently supports three `sort` options:

`'a'`	— sort **alphabetically**
`'n'`	— sort **numerically**
`'f'`	— sort based on **frequency** of occurrence, with most frequent at the top

The default is alphabetic sorting. A good use case for frequency is the case of automobile models, where we might want a Ford "*Mustang*" to appear in the dropdown list before a Lamborghini "*Murcielago*".

Click to show all option. When `withClickToShowAll` is set to `true`, the suggest list functions as a hybrid suggest list—dropdown list component. A down-arrow button appears to the right of the input box and clicking on this button presents the user with the complete list of options in a dropdown list (*fig. 11*). This option is appropriate for static option lists where the number of entries is not very great (*e.g., less than 20*).

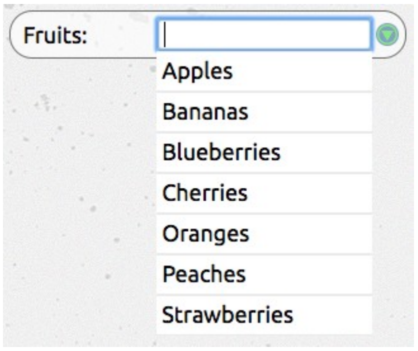


Fig. 11. Hybrid suggest—dropdown list with dropdown button.

Automatic repositioning of dropdown list. Sometimes an input box will appear at the bottom or corner of the browser's viewport. In such cases, if there is not enough room to display the dropdown list below the input box, `Suggest` will try to place the list above the input box or to the right of the input box. Occasionally `Suggest` may need to place the list just to the left of the input box. In this way, to the extent possible, entries in the suggest list remain in full view (*fig. 12*).

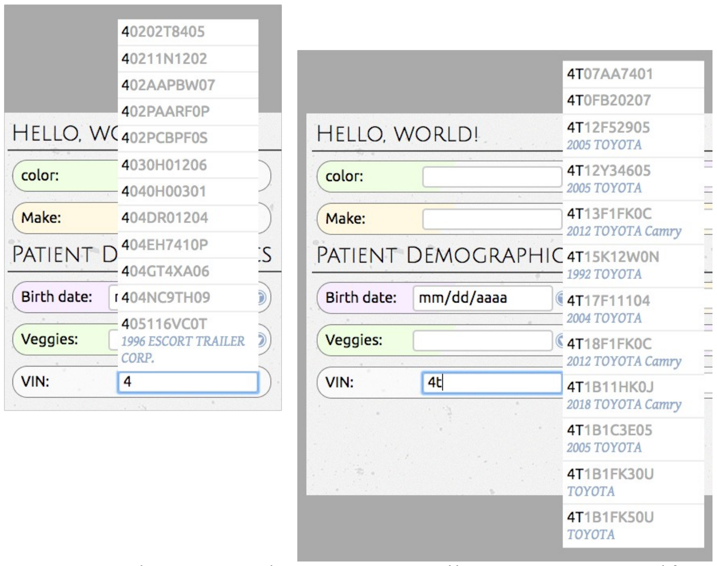


Fig. 12. The suggest list automatically repositions itself to keep as many entries in the list in view as possible.

Input.Suggest Data Sources & Fetch Data Factories

The suggest list requires a list of items against which it matches user input. This list may be a static list of data, or a dynamic list provides by a URL to some data resource. Regardless of the source of data, `v3.input.Suggest` handles data in a uniform manner which simplifies various use cases.

Required Data: At a minimum, Each entry in a list passed to `Suggest` requires an *item*. You may additionally choose to provide a description, *desc*, and an *image* to accompany the item. *Items* must be aggregated into an *array*.

Here is a minimal array in which the entries only have the *item* property present:

```
const fruits = [{item:'apples'},{item:'bananas',{item:'pears'};
```

In contrast, here is an array of U.S. states where `item, desc`, and `image` are all present:

```
const states=[
      {item:'AL',desc:'Alabama',img:'images/AL.svg'}
      {item:'AK',desc:'Alaska',img:'images/AK.svg'},
            .
            .
            .
      {item:'WY',desc:'Wyoming',img:'images/WY.svg'}
];
```

Here is what the states `Suggest` list looks like (*fig. 13*):

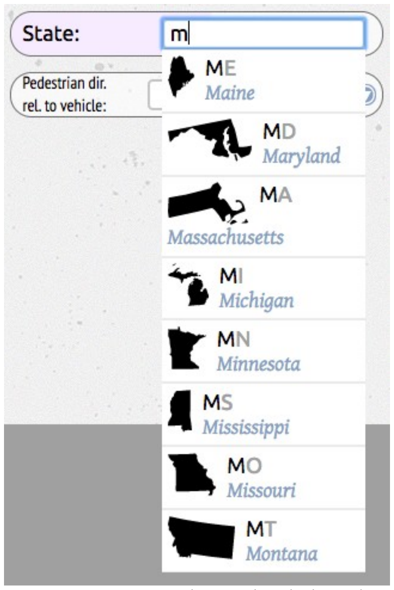


Fig. 13. A Suggest list which has both descriptions and images.

fetchData Method: A `Suggest` list cannot consume a static list such as the `fruits` list or `states` list above directly. Instead, you must provide `Suggest` with a `fetchData` method. This is true even for the case of static data. The primary reason for this is to provide a single uniform interface for handling both static and dynamic data.

fetchDataFactory: Although sometimes you will need or want to write the required `fetchData` method directly yourself, for convenience we also provide a factory method which will return a `fetchData` method in the correct format required by Suggest when provided with a simple array of items:

```
      const getFruit = v3.util.makeFetchDataFactory([
            'Apples','Bananas','Cherries','Strawberries',
            'Oranges','Blueberries','Peaches'
      ]);
```

Note that `makeFetchDataFactory` is only useful for simple static arrays of text strings. For dynamic data, see `makeFetchRemoteDataFactory` below.

Dynamic Data: In contrast, here is a sample `fetchData` method to obtain vehicle VIN numbers. As you can see, the function signature has two parameters: 1) a query string, *v* and 2) a *callback* parameter; additionally, a *url* to a data resource must be provided:

```
function getVIN(v,callback){
    const url = "https://icamlinux.surg.med.umich.edu:4443/vin/?q=" + v ;
    d3.json(url, json => {
      if(!json){
        json={response:[
            {item:"ERROR",desc:"Network error",img:"images/error.svg"}
        ]};
      }
      const response = json.response;
      callback(response);
    });
  }
```

The `Suggest` list instance will pass the query string *v* along with its own data handler as the *callback* parameter.

The URL needs to return data in the JSON format shown for the `states` data above. For examples of how the data server needs to be written, see:

```
      /var/www/html/node_server/node_server2.js
```

... on the icamlinux server.

FetchRemoteDataFactory. V3 also includes a data factory for fetching remote data. This simplifies creation of suggest lists in the general case. `v3.util.fetchRemoteDataFactory` takes up to four parameters:

1) `c:` column to search
2) `t:` table to query. Be sure to include schema prefix, e.g., *vanessa.sys_lu_vehicles, etc.*
3) `m:` match rule: `'b'` to match from beginning, `'w'` to match from the start of words, `'a'` to match anywhere within the string. If not specified, the default is `‘b’`.
4) `s:` sort order: `'a'` for alphabetic, `'n'` for numeric, and `'f'` for frequency descending. If not specified, the default is `'a'` for alphabetic.
5) `n:` limit the number of items returned to *n*. If not specified, the default is 12.

Omission of the `m,` `s` and `n` parameters is possible and will result in the defaults (*match from beginning; sort alphabetically; limit to 12 items returned at one time*) being applied.

`FetchRemoteDataFactory` returns a function having the following signature which is required by `v3.input.suggest`:

```
      foo(v,callback)
```

Here is example usage:

```
// Make a remote data factory for pedestrian kinematics:
const getPedestrianKinematics = v3.util.makeFetchRemoteDataFactory(
      'pedestrian_kinematics',
      'vipa.cases_2018_01_12',
      ‘b’,
      ’a’
);
// Create the data entry component:
const pedKin  = new v3.input.Suggest({
      column:'pedestrian_kinematics',
      parent:container,
      fetchData:getPedestrianKinematics,
      about:'Enter pedestrian kinematics’
});
```

Here is the result of the above code (*fig. 14*):

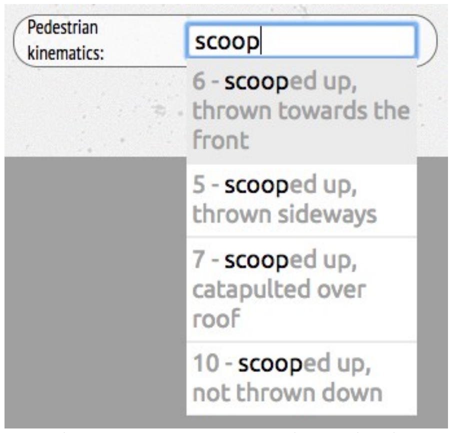

Pedestrian kinematics:
scoop
6 - scooped up, thrown towards the front
5 - scooped up, thrown sideways
7 - scooped up, catapulted over roof
10 - scooped up, not thrown down

Fig. 14. Pedestrian kinematics suggest list which uses a fetchData method created by v3.util.fetchRemoteDataFactory

Allowing Description Searches: Sometimes we want to be able to type in a text string in order to obtain a code value. Common use cases include searching for an AIS code or searching for an ICD9 or ICD10 code. For example, we might want to enter the word "*femur*" in order to retrieve a short list of all AIS or ICD9 or ICD10 codes that contain the word *femur*.

In this case the `item` property would contain a code number, such as "851808"; and the `desc` property would contain a description, such as "femur fracture, head."

To implement a suggest list for this kind of case, you have to do two things. First, you need to write the server-side data handler. The AIS case provides a good example and may be found at `/var/www/html/node_server/node_server2.js` mentioned previously.

Secondly, you must set `allowDescSearch=true` when constructing the suggest list:

```
const aisWidget = new v3.input.Suggest({
      allowDescSearch:true, // let v3 know that matches may occur on descriptions
      tag:'common',         // field common to both ICAM and VIPA
      column:'ais',         // populate the AIS column in the database
      parent:container,     // parental DOM node
      label:'AIS',          // label
      fetchData:getAIS,     // fetchData method
      about:'Enter AIS code –or– description keywords' // tooltip help text
});
```

If you neglect to set `allowDescSearch=true`, the network calls and responses will appear correct, but the dropdown list will never appear.

Here is the AIS suggest list in action (*fig. 15*):

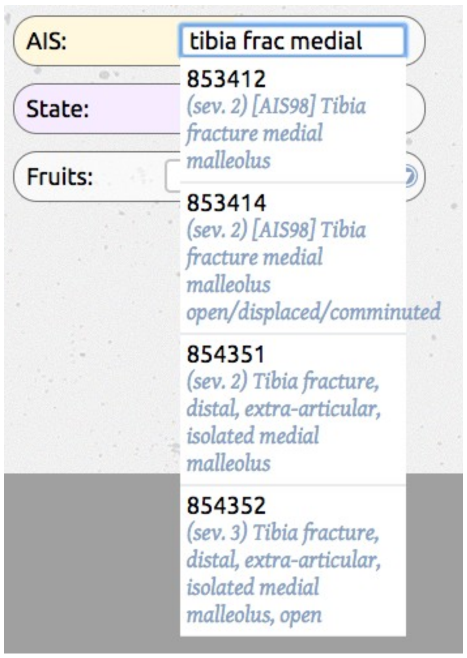

AIS:
tibia frac medial
State:
Fruits:
853412
(sev. 2) [AIS98] Tibia fracture medial malleolus
853414
(sev. 2) [AIS98] Tibia fracture medial malleolus open/displaced/comminuted
854351
(sev. 2) Tibia fracture, distal, extra-articular, isolated medial malleolus
854352
(sev. 3) Tibia fracture, distal, extra-articular, isolated medial malleolus, open

Fig. 15. AIS suggest list which allows searching on terms found in the description column.

Filtering suggest list response data based on values in another input box. The input boxes themselves have their `ids` set to the `column` names. Therefore, one can filter the response data of one suggest list based on values in one or more other suggest lists or other input components present on a form. For example, the *vehicle model* suggest list is automatically filtered if a value is already present in the *vehicle make* suggest list. For an example of how this is done, see the `getVehicleModel` custom fetchData method and corresponding data handler on `node_server2.js`.

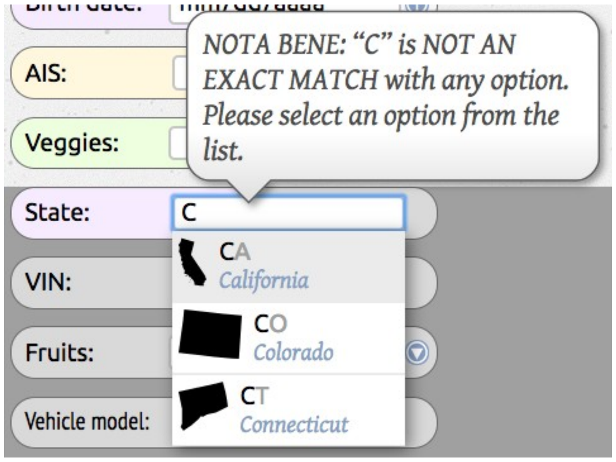
Warning levels on unmatched entries. A user may enter a value in a suggest list that does not *exactly match* to any suggestions in the dropdown list.

If there is a *partial match* and the user presses *ENTER*, then the *first option* in the list will be entered in the input box. If there is *no match* and the user presses *ENTER*, then nothing happens. This behavior already provides a modicum of control.

However, the user can still press the *TAB* key. By default, pressing the *TAB* key leaves the current value in the input box unchanged. Therefore, additional checking of entered values can be performed when a user tries to *TAB* out of an entry field. `Suggest` supports three `warnLevels` on such cases as follows:

1) `'none'` — Don't warn. Allow any entry.
2) `'allow'` — Warn the user about a non-matching entry, but ultimately allow it.
3) `'disallow'` — Warn the user about a non-matching entry and disallow it.

Here is an example where `warnLevel='disallow'` (fig. 16):



NOTA BENE: "C" is NOT AN EXACT MATCH with any option. Please select an option from the list.

Fig. 16. A suggest list constructed with warnLevel='disallow'.

Limiting the number of items returned. By default, the `makeFetchRemoteDataFactory` method will set the limit on the number of items returned to 12. However, this can be changed to accomodate a larger number if needed. Refer to the section on `makeFetchRemoteDataFactory` for details.

Currently available remote data responders. This is a list of data responders currently available on the node server (`node_server2.js`):

`https://icamlinux.surg.med.umich.edu:4443/ais/?q=<code or description words>`

`https://icamlinux.surg.med.umich.edu:4443/suggest_list?q=<query_term>&c=<column>&t=<table>&m=<match rule>&s=<sort order>`

`https://icamlinux.surg.med.umich.edu:4443/vehicle_model?q=<vehicle_model>&m=<vehicle_make>`

`https://icamlinux.surg.med.umich.edu:4443/vin/?q=<code>`

Creating a Form from a Data Dictionary

NOTA BENE: This section describes work that is still in progress. Therefore, not all details of the implementation have been worked out yet. Various details may change over time or may differ from what is written here.

The Vanessa database now contains a system lookup table, `sys_lu_data_dictionary` with the following fields:

```
    Column      |  Type   |   Description
---------------+---------+----------------------------------------------------------
 id            | integer | not null default nextval('sys_lu_data_dictionary_id_seq'::regclass)
 section_table | text    | name of the database table to which the column belongs
 table_column  | text    | a column in the table
 label         | text    | label to use on user interface component
 description   | text    | descriptive text to use on the ‘about’ tooltip
 ui_type       | text    | type of component (basic, number, date, suggest_dropdown, or suggest)
 options       | text    | array of item options for a static suggest_dropdown
 ui_order      | integer | order in which the ui component should appear on the form
 tag           | text    | tag (‘vipa’,’common’,’phi’, etc.)
 fetch_data    | text    | column for specifying custom fetch data methods
 match         | text    | the match rule (one of ‘b’,’w’, ‘a’; or NULL)
 sort          | text    | the sort order (one of ‘a’,’n’,’f’; or NULL)
 warn_level    | text    | warning level (one of ‘none’,’allow’,‘disallow’,NULL)
 max_items     | integer | specify a limit to the number of items returned
```

The idea is to use this table to serve as the data source for the construction of user-interface forms within the Vanessa application.

In its current form, this table is sufficient for a proof-of-concept demonstration of form creation, described below.

The node server `node_server2.js` provides a url to retrieve the data dictionary in json form:

    https://icamlinux.surg.med.umich.edu:4443/get_dd

As a reference, it is worth repeating here briefly the meanings of the match, sort, and warn_level flags:

Match Rules. The match rule flags are:

b	— match from the *beginning* of the string. This is also the *default*.
w	— match from the beginning of *words*
a	— match *anywhere*

Sort Order. The sort order flags are:

a	— *alphanumeric* sort. This is also the *default*.
n	— *numeric* sort.
f	— *frequency* of occurrence sort.

Warning Levels. The warn_level flags are as follows:

none	— No restrictions are imposed on what the user can enter.
allow	— Warn on non-matching entries, but allow them.
disallow	— Warn on non-matching entries, and disallow them.

Form Creation Pseudocode

```
json = retrieve the data dictionary from database as a json array of row objects;
const response = json.response;
response.forEach( row => {
      switch(row.ui_type){
      case ‘date’:
            create a new v3.input.Date with relevant row data;
            break;
      case ‘number’:
            create a new v3.input.Number with relevant row data;
            break;
      case ‘suggest_dropdown:
            if(row.match is null){ set row.match to the default; }
            if(row.warn_level is null){ set row.warn_level to the default; }
            create a new v3.input.Suggest with relevant row.options
                  and withClickToShowAll:true;
            break;
      case ‘suggest’:
            if(row.match is null){ set row.match to the default (string start); }
            if(row.sort is null){ set row.sort to the default (alphabetic); }
            if(row.warn_level is null){ set row.warn_level to the default (none); }
            if( there is a custom row.fetch_data){
                  use the specified custom row.fetch_data method;
            }else{
                  use v3.util.makeFetchRemoteDataFactory
                        to make a fetchData method;
            }
            create a new v3.input.Suggest
                  with relevant row options and fetchData method;
            break;
      default:
            create a new v3.input.Basic component
                  with relevant row options;
            break;
      }
});
```

*** END OF DOCUMENT ***