Data Management Apps with Rebol

A shortened tutorial which explains only the basics needed to build useful CRUD (create read update delete) data management apps with Rebol.

By: Nick Antonaccio

1. Getting Started
   - 1.1 Download
   - 1.2 Functions and Word Labels
   - 1.3 Data Types
   - 1.4 GUI Basics
2. A Few Short Apps
   - 2.1 How to Create Distributable Apps
   - 2.2 Dinner Tip Calculator
   - 2.3 Days Between 2 Dates
   - 2.4 Notepad
3. More About Functions
   - 3.1 User Created Functions
4. Blocks of Data
   - 4.1 Block Functions
   - 4.2 Foreach
5. Flow Control (conditional evaluations and loops)
6. More Example Apps
   - 6.1 To-Do List
   - 6.2 Personal Scheduler
7. Generic CRUD App with Form and Data Grid Displays
8. Grids with the 'List' Widget
   - 8.1 Display Tens of Millions of Lines, With Fast Performance
   - 8.2 A Practical Set of Features
   - 8.3 Cash Register App Using the Basic Grid Features
   - 8.4 Even More Features
9. Parse
10. Some Additional App Examples to Study
   - 10.1 Cash Register
   - 10.2 Cash Register Report
   - 10.3 Web Chat
   - 10.4 Simple File Sharer
   - 10.5 Simple Full Screen Slide Presentation
   - 10.6 Simple Encrypt/Decrypt Utility
   - 10.7 More Advanced To-Do List
   - 10.8 Recursive Text in Files Search
   - 10.9 Little Blogger
   - 10.10 Console Calendar Printer
   - 10.11 Continuously Echo Clipboard to File
   - 10.12 FTP Tool
   - 10.13 More GUI Examples
   - 10.14 Network Apps (HTML Server and Handler, Text Messagers, Email, VOIP, Etc.)
   - 10.15 Web Site (CGI) App Examples
   - 10.16 DLL/Shared Lib Examples
   - 10.17 Graphics Examples
   - 10.18 Sound Apps
   - 10.19 Built-In Help and Reference Materials
11. Learning More

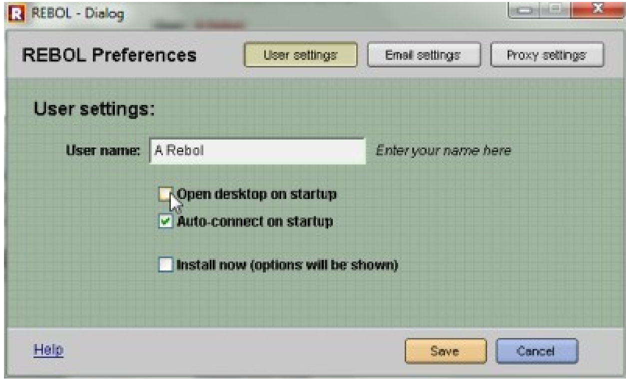
1. Getting Started

1.1 Download

Download Rebol/View from http://www.rebol.com/download-view.html (it's tiny, about 1/2 megabyte (~*500k*) on most platforms).

Run Rebol, click the "Console" icon, and paste in any code example in this text.

(To make Rebol automatically open to the console every time it starts, click the 'User' desktop menu, then uncheck 'Open desktop on startup').



1.2 Functions and Word Labels

Function words perform some action upon data 'argument' values. In the example below, 'notify' is the function and the text "Hello World!" is the data argument. The 'notify' function displays its text argument in a pop-up dialogue box. Paste this and all following examples into the Rebol interpreter, to see what the code does:

```
notify "Hello World!"
```

Word labels allow you to give names to data values. In Rebol, word labels are set using the *colon* symbol:

```
name: "John"
notify name
```

The data value represented by a word label can be, for example, a number, a quoted string of characters, a date, a color value, or even a large database of information, the content of a complete web site, an entire book of text, the code of multiple executable Rebol programs, etc.

Capitalization in word labels doesn't matter in Rebol:

```
name: "Joe"
Name: "Bob"
NAME: "Bill"
```

You can join data values together using the 'rejoin' function:

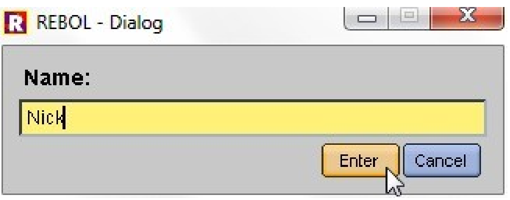
```
name: "John"
message: rejoin ["Hi " name "!"]
notify message
```

Most functions 'return' (output) data values. Notice in the example above that the return value of the 'rejoin' function is assigned the label 'message'. The 'request-text' function below returns the text which the user types into a pop-up text requestor dialogue. Here, the return value of the 'request-text' function is assigned the label 'name':

```
name: request-text
message: rejoin ["Hi " name "!"]
notify message
```

Many functions have available '/refinements' (options) which adjust how the functions operate. The '/title' refinement of the 'request-text' function allows you to set the displayed title text:

```
name: request-text/title "Name:"
message: rejoin ["Hi " name "!"]
notify message
```



You can use the *return value of one function as the data argument of another function*. In this example, the return value of the 'request-text' function is used as a data value argument of the 'rejoin' function, and the return value of the 'rejoin' function is used as the argument of the 'notify' function. So, in order for this code to complete, the user must respond to the 'request-text' pop-up near the end of the line - it's output is needed as a data value, before either of the other functions can complete their actions (notify waits for rejoin to return a value, rejoin waits for request-text to return a value):

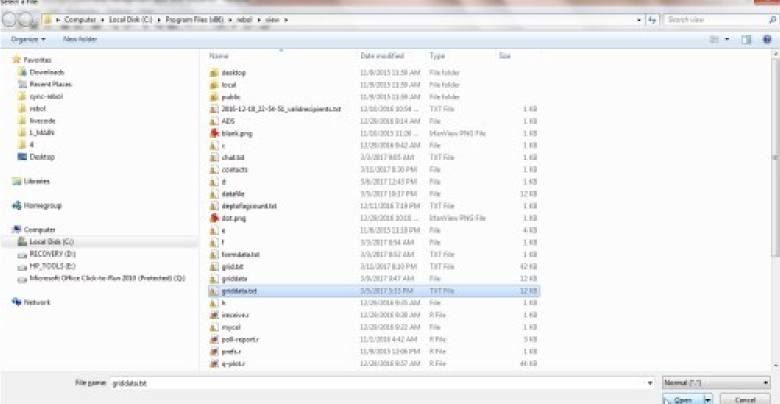
```
notify rejoin ["Hi " request-text/title "Name:" "!"]
```

Here, the return value of the 'now' function is used as the argument of the 'form' function, and the return value of the 'form' function is used as the argument of the 'notify' function (the 'form' function converts the date value into a string of text which the 'notify' function can display):

```
notify form now
```

The functions below are 'editor' and 'request-file' (the '/only' refinement of the 'request-text' function limits the user's file selection to a single file). The first thing the user sees here is the request-file dialogue, because the return value of that function is needed for the editor function to complete its action:

```
editor request-file/only
```



1.3 Data Types

Rebol has many more built-in data types than most languages (~40): integers, decimals, text strings, money values, dates, times, colors, coordinate pairs, images, URLs, etc. Rebol automatically knows how to perform appropriate computations with known data type values. Note that in these examples, any 'comments' following a semi-colon are ignored by Rebol:

```
6:30am + 00:37:19     ; Rebol adds/subtracts time and date values correctly
4-may-2017 - 45
now + 0:0:59
now - 45

$29.99 / 7                              ; money computations are rounded properly
29.99 / 7            ; (whereas this number has many digits after the decimal)

255.0.0         ; RGB colors are Red.Green.Blue values limited to 255.255.255
red / 2                  ; you can perform computations upon color values too

23x54 + 19x31                                ; math with coordinate pairs is easy
22x66 * 2                     ; (this is useful in apps that deal with graphics)
22x66 * 2x3
```

You can convert ("cast") values to different types:

```
to-integer "5"                ; this converts a string to an integer value
5 + 6                         ; you can perform math operations with integers
"5" + "6"                     ; (error) you can't perform math with strings
(to-integer "5") + (to-integer "6")    ; this makes the math operation work
(do "5") + (do "6")           ; 'do' can convert strings to known data types
(load "5") + load "6"         ; 'load' also works to convert to known types
form http://rebol.com         ; 'form' is like 'to-string', a URL isn't text
notify http://msn.com         ; (error) 'notify' needs a text value argument
notify form http://msn.com    ; this gives the 'notify' function a text value
```

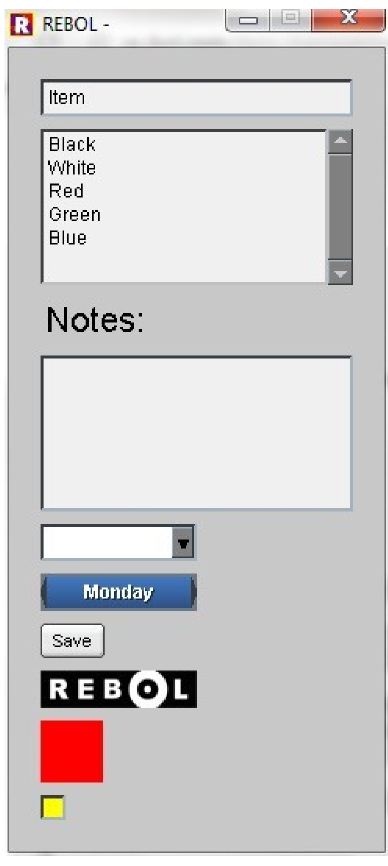
You can generate random values of any type:

```
random/seed now         ; this line is required to generate random values
random 100              ; generates a random number between 1 and 100
50 + random 50          ; a random number between 50 and 100
random 255.255.255      ; a random color
random "abcd"           ; a random arrangement of characters
random ["joe" "bob"]    ; a random arrangement of items
```

1.4 GUI Basics

'GUI' stands for 'Graphic User Interface'. User interfaces are windows (screens layouts) which display buttons, fields, drop-down boxes, etc., that the user of an app can interact with. Try pasting each of these examples into the Rebol console to see the resulting screen layouts:

```
view layout [btn]                       ; this layout displays a button

view layout [btn "Cick Me"]             ; button with some text on its face

view layout [                           ; 2 buttons
  btn "Cick Me First"
  btn "Cick Me Second"
]

view layout [                           ; changing the order of widgets in
    btn "Cick Me Second"                ; code changes their order on screen
    btn "Cick Me First"
]

view layout [field]                     ; displays a text entry field

view layout [field "Type here..."]      ; a field with some default text

view layout [
    field 400 "Name"                    ; different sized fields
    field 400 "Address"
    field "Phone"
   btn "Save"
]

view layout [                           ; some more types of widgets:
    field "Item"
    text-list data ["Black" "White" "Red" "Green" "Blue"] 200x100
    text "Notes:" font-size 22
    area 200x100
    drop-down data ["Bob" "Tom" "Bill"]
    rotary data system/locale/days
    btn "Save"
    image logo.gif
    box 40x40 red                       ; notice colors, sizes, font size...
    check yellow
]
```




```
view layout [
    across                              ; layout widgets across the screen
    btn "Save"
    btn "Load"
    btn "New"
    below                               ; layout widgets below one another
    field
    text "Description:"
    area
]
                                        ; the exact same program as above:
view layout [
    across btn"Save"btn"Load"btn"New"below field text"Description:"area
]
                                        ; (notice that quotes don't need to
                                        ;  be separated by white space)

view layout [
    style b box blue 70x70 "I'm a blue box"    ; 'style' creates a new widget
    across                                     ; based on an existing widget.
    b b b b return                             ; 'return' starts a new line
    b b b b return
    b b b b return                             ; a bunch of 'b' widgets
]
```

Functions can be enclosed in square brackets after the code for any widget in a GUI layout, and those actions will be executed whenever the GUI widget is activated by the user (i.e., whenever the user clicks a button, enters text into a text field, selects an item from a drop-down list, etc.):

```
view layout [
    btn "Cick Me" [notify "Clicked!"]                          ; button click runs
]                                                              ; the 'notify' action
```

The word 'value' is used to refer to the main data displayed by or selected in a widget:

```
view layout [
    field "Type here then press ENTER" [notify value]
    text-list data ["Bob" "Tom" "Jim"] [notify value]
    text "http://yahoo.com" [browse value]
]
```

You can use and change the data values displayed by a widget, by setting the /text property of the widget:

```
view layout [
    f: field
    btn "Submit" [notify f/text]                ; 'f/text' refers to the text
]                                               ; in the field labeled 'f'

view layout [
    f: field
    btn "Submit" [f/text: "Hello!"  show f]     ; sets the text in the f field
]

view layout [
    f1: field
    f2: field
    f3: field
    btn "Submit" [
       f1/text: "Name"     show f1              ; 'show' updates the display
       f2/text: "Address"  show f2              ; of any changed widget data
       f3/text: "Notes"    show f3
    ]
]

view g: layout [
    f1: field
    f2: field
    f3: field
    btn "Submit" [
        f1/text: "Name"
        f2/text: "Address"
        f3/text: "Notes"
        show g                                  ; you can update the entire
    ]                                           ; layout (labeled 'g' above)
]

view layout [
    text "Type your name:"
    f: field
    btn "Submit" [
      notify rejoin ["Hi " f/text "!"]          ; this 'rejoin' combines text
    ]                                           ; from the field labeled 'f'
]

view layout [
    f1: field "Name"
    f2: field "Address"
    btn "Submit" [
      notify rejoin ["You entered: " f1/text ", " f2/text]
    ]
]
```

```
view layout [
  x: field "5"                 ; fields hold ONLY text values - convert them
  y: field "7"                 ; to use in computations
  btn "Compute" [print (to-integer x/text) * (to-integer y/text)]
]

view layout [
    size 600x400                           ; manually set screen size
    at 300x200 btn                         ; set btn location coordinate
]
```

One of the great things about Rebol is that you don't need to use a heavy IDE or GUI builder to create useful screen layouts. You'll find that creating layouts with Rebol code is actually much faster and more convenient than having to drag-and-drop/arrange widgets on screen using the typical bulky IDEs which other languages require. Rebol GUI code can be easily generated by other Rebol code, which makes manipulating screen layouts at run time a simple process, even when dealing with user interface requirements that are complicated to manage in other software development systems.

2. A Few Short Apps

2.1 How to Create Distributable Apps

Type 'editor none' into the Rebol console to bring up Rebol's built in text editor. Press the F5 key (or CTRL + E) in the editor to save and run any edited code. You can use any other text editor you want (Notepad++, Textmate, etc.). Just save your code to a file name ending in '**.r**'. Once installed, the Rebol interpreter automatically opens (runs) any file with a .r extension.

To distribute apps, just have your users install the Rebol interpreter, then click the file icon of your code file. Rebol is extremely small to download (*really tiny - 500k*), but some users still may not want to install it. If that's the case, they can simply drag-drop your downloaded script file onto the downloaded Rebol interpreter, and it will run.

If you want to take a few extra minutes to save your users from having to download Rebol separately, you can package the Rebol interpreter with script file(s) to create small stand-alone executable apps (.exe files) which run just like native compiled apps.

If you want to use the Rebol editor to edit programs, try typing 'do http://re-bol.com/ed.r' into the console to add some essential features to it (undo, redo, etc.).

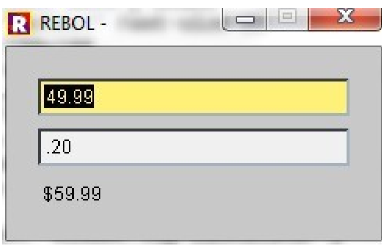
Every Rebol program starts with a header:

```
REBOL []
```

Optional title text and other info about your app goes there:

```
REBOL [
    title: "My App"
    date:  1-feb-2017
]
```

2.2 Dinner Tip Calculator



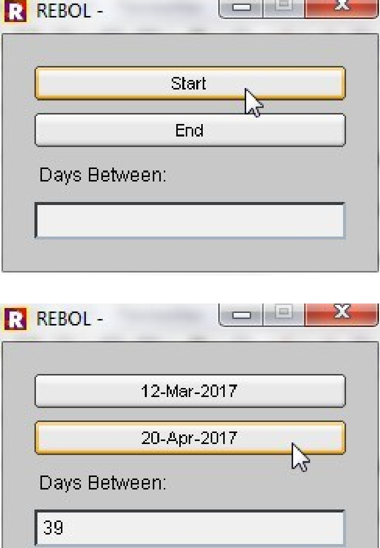
Try pasting this app into the Rebol text editor, or use any other editor. Save the code to a file name ending in '.r', then run it. The app allows its user to enter a total restaurant bill amount and a tip percentage, and it calculates the total which should be paid:

```
REBOL [title: "Dinner Tip Calculator"]
view layout [
     f: field "49.99"
     t: field ".20" [
         x/text: (1 + to-decimal t/text) * to-money f/text  show x
     ]
     x: text "Total, with tip"
]
```

Here's an explanation of the code above, line by line:

1. The first line is the header, with a title. That title text appears in the title bar at the top of the app's GUI window layout.
2. The second line starts a GUI window layout block.
3. A field widget with the default text "49.99" is labeled 'f'
4. A field widget with the default text ".20" is labeled 't'. When that widget's action is activated (when the user enters text into it):
    - The displayed value in the 'x' widget is set to a calculation. That calculation is 1 plus the decimal value displayed in the 't' field, times the money value displayed in the 'f' field. The changed display of 'x' field is then updated with the 'show' function.
5. A text widget with the default text "Total, with tip" is labeled 'x'.

2.3 Days Between 2 Dates



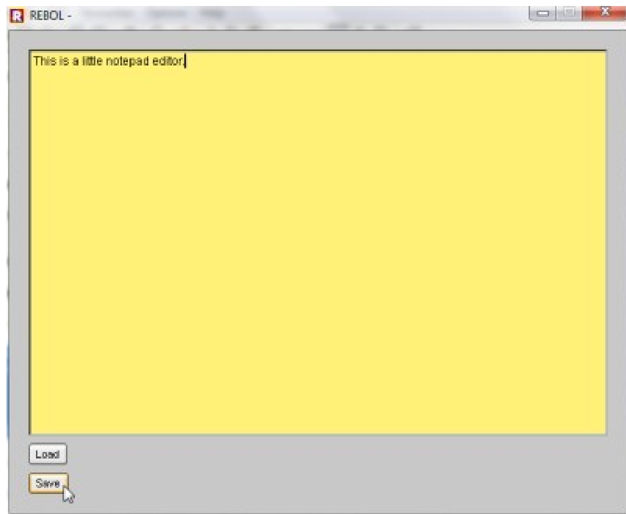
Now take a look at the following app. It allows the user to select any 2 dates, and displays the number of days between those 2 dates:

```
REBOL [title: "Compute Days Between 2 Selected Dates"]
view layout [
    btn 200 "Start" [face/text: s: request-date]
    btn 200 "End" [
        face/text: e: request-date
        f/text: e - s
        show f
    ]
    text "Days Between:"
    f: field
]
```

Here's what's happening in the code above:

1. The first line is the header, with a title. That title appears in the title bar of the GUI window.
2. The second line starts a GUI window layout block.
3. A button widget with the default text "Start" is added to the layout. When the button is pressed by the user:
    - The text on the face of the button is set to the data represented by the word label 's', which is set to the date which the user selects from a pop-up date requestor dialogue.
4. A button widget with the default text "End" is added to the layout. When the button is pressed by the user:
    - The text on the face of the button is set to the data represented by the word label 'e', which is set to the date which the user selects from a pop-up date requestor dialogue.
    - The text of the field labeled 'f' is set to a calculation. That calculation is the difference between the 2 dates selected by the user (the value represented by 'e', minus the value represented by 's').
    - The display of the 'f' field is updated with the 'show' function.
5. A text widget with the default text "Days Between:" is added to the GUI layout.
6. A field widget labeled 'f' is added to the layout

2.4 Notepad



This app allows the user to create, load, save, and edit text files:

```
REBOL [title: "Notepad"]                    ; (tiny text editor)
view layout [
     a: area 600x400
     btn "Load" [a/text: read request-file/only  show a]
```

```
    btn "Save" [write request-file/only a/text  notify "Saved"]
]
```

1. The first line is the header, with a title. That title appears in the title bar of the GUI window.
2. The second line starts a GUI window layout block.
3. A text area widget 600x400 pixels in size labeled 'a'.
4. A button widget with the text "Load" is added to the layout. When the button is pressed by the user:
   - The text on the face of the area widget is set to the data read from a file selected by the user, from a pop-up file requestor dialogue.
   - The display of the 'a' area is updated with the 'show' function.
5. A button widget with the text "Save" is added to the layout. When the button is pressed by the user:
   - The text on the face of the area widget is written to a file selected by the user, from a pop-up file requestor dialogue.
   - The user is notified with a pop-up messaged displaying the text "Saved".

You may notice that if you cancel the process of selecting a file in the app above, the program crashes because the read and write functions have no file which they can use to complete their intended actions. In order to handle this error, we'll wrap each button action block in an 'attempt' block. The 'attempt' function keeps the program from crashing if ever an error occurs in the enclosed block:

```
REBOL [title: "Notepad"]
view layout [
    a: area 600x400
    btn "Load" [attempt [a/text: read request-file/only  show a]]
    btn "Save" [attempt [write request-file/only a/text  notify "Saved"]]
]
```

3. More About Functions

Rebol has hundreds of built-in functions which do useful things. The sections below demonstrate the function words you'll use most. Notice that the following 'request' function makes use of a string of data several lines long, which is enclosed in *curly braces*:

```
request {Functions perform ACTIONS.  Parameters are DATA.  Make sense?
  Remember that most functions RETURN a useful value...  Notice in this
  example that multiple lines of text are enclosed in CURLY BRACES,
  instead of quotes.}
```

Notice the use of the '%' symbol when reading, writing and editing files

```
write %temp.txt "asdf" ; write takes 2 arguments, a file and data to write
editor %temp.txt    ; notice that FILE NAMES start with the PERCENT SYMBOL
mytext: read %temp.txt
write/append %temp2.txt ""   ; create file (or if it exists, do nothing)
```

These requestor functions don't require any parameters, but they RETURN (output) a useful value, which can be assigned a word label:

```
request-text
request-date
request-color
request-file
ask "Enter your name:  "  ; 'ask' gets console text input from user, no GUI
```

Many functions have available 'refinements' (options) which adjust how they operate:

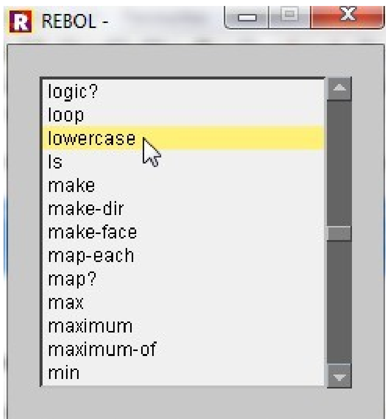
```
request-text/default "Text" ; some default text displayed in the requestor
request-text/title/default  "Name:"  "John Smith"     ; 2 options together
editor request-file/only   ; the 'only' refinement limits choice to 1 file
call/show "notepad.exe c:\config.sys"   ; run and show an OS shell command
```

Composing (combining) functions together is very important in Rebol:

```
print read %temp.txt  ; RETURN value of 2nd func is ARGUMENT of 1st func
print trim/lines {    The 'trim' function trims extra space from
  strings of text.  It has many refinements to help trim space in common
  ways.  Try typing 'help trim' into the Rebol console.}
editor request-file/only  ; 1st func runs only AFTER 2nd func completes
notify rejoin ["You typed: " request-text] ; request-text > rejoin > notify
write clipboard:// (read http://rebol.com)  ; 2nd parameter in parentheses
editor clipboard://   ; try opening notepad and pasting from the clipboard
print read %./               ; print file names read from current folder
```

Run this GUI script to list all the functions built into Rebol, and click any word to get help:

```
w: copy []  foreach i copy first system/words [
  attempt [if any-function? get to-word i [append w i]]
] view layout[text-list data sort w [x: to-word value ? :x  print "^/^/"]]
```



```
REFINEMENTS:
     /part -- Limits to a given length or position.
         range -- (Type: integer any-string)


USAGE:
    LOWERCASE string /part range

DESCRIPTION:
     Converts string of characters to lowercase.
     LOWERCASE is a native value.

ARGUMENTS:
     string -- (Type: any-string char)

REFINEMENTS:
     /part -- Limits to a given length or position.
         range -- (Type: integer any-string)
```

3.1 User Created Functions

You can create your own functions, which allow you to group any number of actions under a single word label that you assign. Functions also allow you to include 'arguments' or labeled data values which are processed by the function.

In this example, a function labeled 'saymantra' is created. It accepts one argument, labeled 'arg1'. Whenever the function is run, it notifies the 'arg1' text 3 times. In the example below, the 'saymantra' function is executed whenever the user clicks the button in the GUI layout:

```
saymantra: func [arg1] [
    notify form arg1
    notify form arg1
    notify form arg1
]
view layout [
    btn "Repeat Mantra" [
        saymantra {I am good enough, strong enough, and gosh darnit...}
    ]
]
```

The 'does' structure is a shortcut for creating functions which don't require an argument value:

```
cls: does [print newpage]
cls
```

You can also use the 'do' function to execute blocks of code (block structures are covered in the next section):

```
cls: [print newpage]
do cls
```

4. Blocks of Data

4.1 Block Functions

In Rebol, lists and tables of data values are stored in 'blocks'. Blocks are represented in code by surrounding a group of data values with square brackets. Data values in blocks are typically separated by white space (space characters, tabs, new lines, etc.):

```
names: ["John" "Joe" "Bill" "Bob" "Dave"]
```

The 'probe', 'print', and 'editor' functions are useful for examining blocks of data:

```
names: ["John" "Joe" "Bill" "Bob" "Dave"]
probe names
print names
editor names
```

Use the 'append' and 'insert' functions to add values to a block:

```
names: ["John" "Joe" "Bill" "Bob" "Dave"]
append names "Mike"                        ; adds to the end of block
print names
insert names "Tom"                         ; adds to the beginning of block
print names
insert at names 2 "George"                 ; adds at second position in block
print names
```

Create a empty data block by assigning a word label to empty square brackets. Use the 'copy' function to ensure that the word is assigned a fresh new block:

```
mythings: copy []
probe mythings
append mythings "car"
append mythings "boat"
probe mythings
```

There are several different syntax patterns which you can use to pick values from numbered index locations in a block:

```
names: ["John" "Bill" "Tom" "Mike" "George" "David"]

x: 1
y: 2

print first names                     ; 7 different ways to pick the 1st item
print names/1                         ; (these 7 lines all do the exact same
print names/:x                        ;  thing)  'PICK' IS THE MOST VERSATILE
print pick names 1
print pick names x
print compose [names/(x)]
print reduce [names/(x)]

print second names                    ; 7 different ways to pick the 2nd item
print names/2                         ; People tend to prefer one syntax
print names/:y                        ; pattern over another, but it really
print pick names 2                    ; doesn't matter which one you use.
```

```
print pick names y
print compose [names/(y)]
print reduce [names/(y)]
```

You can use the 'find' function to search for data in a block:

```
print find names "Bill"        ; finds Bill in list (returns rest of block)
print find names "Sue"         ; returns 'none' when the value isn't found
print first find names "Bill"  ; returns the first value in rest of block
print index? find names "Bill" ; prints position of Bill in list
print find/last names "Bill"
```

You can remove one or more items from a block:

```
remove names                ; removes 1 item wherever the index is in block
remove find names "Mike"    ; FIND SETS INDEX TO WHERE FOUND ITEM IS LOCATED
remove/part names 2         ; removes 2 items wherever index is
```

You can alter items in a block:

```
change third names "Phil"
poke names 3 "Joe"                  ; same operation as line above
copy/part names 2                   ; gets 2 items from wherever index is
replace names "Joe" "Jim"           ; replaces first occurrence of "Joe"
replace/all names "Jim" "Al"        ; replaces all occurrences of "Jim"
```

You can make a copy of a block when performing changing operations, so that the original block isn't effected:

```
reverse copy names        ; returns reversed COPY of names block
reverse names             ; here, the names block is permanently reversed
```

You can count the number of items in a block (and of course, assign a word label to the result, or use the result as the argument to another function):

```
length? names
sz: length? names
print sz
print length? names
```

You can pick out every x items in a block:

```
extract names 3              ; every third item
extract/index names 3 2      ; every third item, starting with second item
```

You can sort items in a block (or make a copy of the block to sort, without affecting the original block):

```
names: sort copy names             ; "copy" keeps names block from changing
names: sort names                  ; here, names block is permanently sorted
table: [3 "Jim" 1 "John" 2 "Bob"] ; Blocks can contains rows and columns
```

```
sort/skip table 2                  ; SORT TABLES using sort/skip
```

You can combine and compare blocks:

```
names: ["John" "Bill" "Tom" "Mike" "George" "David"]
morenames: ["Jim" "Jeff" "Stan" "Peter"]

join names morenames      ; combines both lists
intersect names morenames
difference names morenames
exclude names morenames
union names morenames
unique names
clear names
empty? names
```

These functions move the index pointer location to a different position in the block:

```
head names     ; sets index (pointer) to beginning of list
next names     ; moves index to next item
back names     ; moves index to previous item
last names     ; moves index to last item
tail names     ; moves index after last item
at names 2     ; moves index to second item
skip names 1   ; moves ahead 1 position
index? names   ; returns current index number
insert (at names 3) "Lee"    ; inserts "Lee" into 3rd position in list
```

The 'form' function is used to convert blocks to strings, and the 'mold' function is used to serialize entire blocks (useful when saving blocks to disk, transferring over a network connection, etc.):

```
probe form names
probe mold names
```

The 'reduce' function is used to get values from nested blocks:

```
fruits: ["apple" "orange" "banana"]
[fruits/1 fruits/2]                ; returns unevaluated items (just words)
reduce [fruits/1 fruits/2]                        ; returns evaluated items
foreach i [fruits/1 fruits/2] [probe i]
foreach i reduce [fruits/1 fruits/2] [probe i]
```

Getting specified values from a block is accomplished using 'copy', 'at', 'find', 'skip', 'index?', 'length?' and related functions, along with index numbers:

```
copy/part names 3
copy/part (at names 4) 3
copy at tail names -3
copy/part (at names 2) 3
copy/part (find names "Jim") -3
copy/part (skip (find names "Jim") -6) 3
notify form (copy/part names 3)
i: ((index? (find names "Stan")) - 1)
print pick names i
l: length? names
print pick names l               ; returns last item
```

```
print pick names (l - 1)          ; returns next to last item
print pick names (random l)       ; returns random item
```

You can move the position of values in a block, with the 'move' function:

```
x: ["red" "green" "blue"]
move/to z: find x "green" (index? z) - 1
```

In Rebol, strings of text are treated as series of characters, and all the block functions can be used to manipulate characters in a string. The # symbol is user to refer to specific characters:

```
find "asdfgh" #"f"

x: "asdfgh"
move/to z: find x #"g" (index? z) - 1
```

4.2 Foreach

The 'foreach' function allows you to perform some action upon each sequential item in a block. If your intent is to create applications which accomplish the same end goals as spreadsheets and tools like MS Access, Filemaker, etc., 'foreach' will be one of your most commonly used tools. You'll use 'foreach' any time you need to deal with lists of data, or tables of information made of rows and columns:

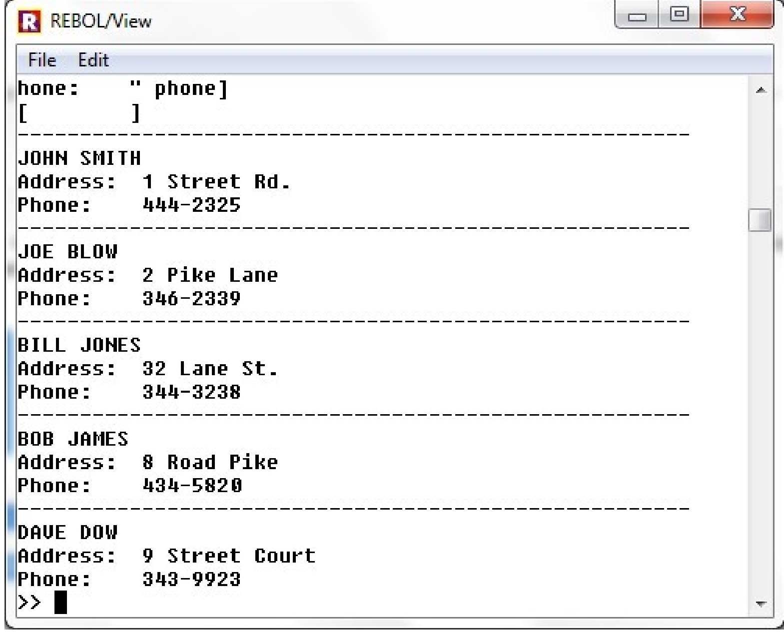
```
names: ["John" "Joe" "Bill" "Bob" "Dave"]
foreach name names [print name]
```

Note that the word 'name' in the foreach function above is an arbitrary word label which you can choose, just like the 'names' block label. The following code does the exact same thing as the example above:

```
x: ["John" "Joe" "Bill" "Bob" "Dave"]
foreach i x [print x]
```

The 'foreach' function can go through blocks of data values in labeled sequential *groups* (rows made of repetitive *columns* or 'fields'), to process each row of a data in a table of information. Here, the 'foreach' function loops through groups of 'name', 'address', and 'phone' values in the 'employees' block:

```
employees: [
    "John Smith"   "1 Street Rd."     "444-2325"
    "Joe Blow"     "2 Pike Lane"      "346-2339"
    "Bill Jones"   "32 Lane St."      "344-3238"
    "Bob James"    "8 Road Pike"      "434-5820"
    "Dave Dow"     "9 Street Court"   "343-9923"
]
foreach [name address phone] employees [
    print "---------------------------------------------------"
    print uppercase name
    print rejoin ["Address:  " address newline "Phone:    " phone]
]
```



The groups of data values above are indented and spaced so that the code is neatly readable by humans, but Rebol doesn't care about that spacing. Each piece of data just needs to be separated by some (any kind of) white space. The 'foreach' function above takes every three consecutive values in the block, and labels them 'name' 'address' and 'phone', then performs the actions between the square brackets upon each of those successive groups (rows) of 3 values.

Blocks can be nested within outer blocks. When creating data structures to store and process tables of data, you can choose to use either simple sequential lists such as the 'employees' block shown above, or you can enclose each row in a separate nested block:

```
employees: [
    ["John Smith"   "1 Street Rd."     "444-2325"]
    ["Joe Blow"     "2 Pike Lane"      "346-2339"]
    ["Bill Jones"   "32 Lane St."      "344-3238"]
    ["Bob James"    "8 Road Pike"      "434-5820"]
    ["Dave Dow"     "9 Street Court"   "343-9923"]
]
foreach row employees [
    print "-----------------------------------------------------"
    print uppercase row/1
    print rejoin ["Address:  " row/2 newline "Phone:    " row/3]
]
```

One benefit of using nested blocks is that you can easily add/remove columns to/from each row record, without any significant changes to the existing data structure or code. If you use sequential lists, you need to be very careful that each group of sequential data values includes the proper number of column values (or the rows will become mis-aligned).

'Foreach' functions can also be nested within 'foreach' functions, as in the example below. Note the use of the 'sort/skip' function in this example, which sorts the table by date before the foreach function runs:

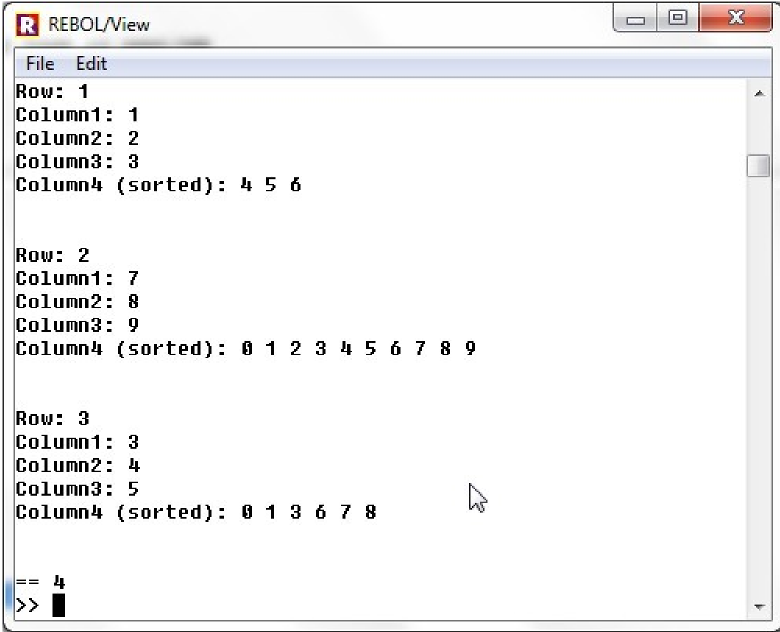
```
appointments: [
    5-may-2017/11:30am   "John Smith"   [%js1.txt %js2.txt]
    16-feb-2017/5:00pm   "Joe Blow"     [%jb1.txt]
    23-apr-2017/1:15pm   "Mary Jones"   [%m1.txt %m2.txt %m3.txt]
]
foreach [date name files] (sort/skip copy appointments 3) [
```

```
    print rejoin [name ", " date ", have these files ready:  ^/" ]
    foreach file files [
        print file
    ]
    print "^/"                          ;  "^/" IS THE SAME AS NEWLINE
]
```

```
Joe Blow, 16-Feb-2017/17:00, have these files ready:

jb1.txt


Mary Jones, 23-Apr-2017/13:15, have these files ready:

m1.txt
m2.txt
m3.txt


John Smith, 5-May-2017/11:30, have these files ready:

js1.txt
js2.txt


>>
>>
>>
>>
>>
```

Take a look at how the 4th item in each line in the block labeled 'data' below, is a nested block of data values (a sub-block):

```
data: [
    1 2 3 [4 5 6]
    7 8 9 [0 9 8 7 6 5 4 3 2 1]
    3 4 5 [6 3 1 7 8 0]
]
probe data/2
probe data/4
probe data/4/2
counter: 1
foreach [col1 col2 col3 col4] data [
    print rejoin [
        "Row: " counter newline
        "Column1: " col1 newline
        "Column2: " col2 newline
        "Column3: " col3 newline
        "Column4 (sorted): " (sort col4) newline newline
    ]
    counter: counter + 1
]
```



You can use foreach loops to deal with folder (file directory) listings on a local hard drive, thumb drive, FTP server, etc.:

```
folder: read %.
foreach file folder [print file]  ; prints every file in current directory
foreach file (read %./) [print file] ; load the dir block directly in loop
foreach file read %./ [print file]          ; parentheses aren't required
foreach month system/locale/months [print month]; this is a built-in block
```

Combining conditional operations (if, either, any, etc.) with loops (foreach, repeat, etc.) is one of the most common things that happens in programming (more about conditional operations will demonstrated in the next section):

```
schedule: ["John" 8:00am "Joe" 9:00am "Bill" 2:00pm "Bob" 6:00pm]
foreach [name time] schedule [print [name time]]
foreach [name time] schedule [
  if time > 12:00 [
    print rejoin [
      name "'s appointment is in the afternoon"
    ]
  ]
]

foreach i fruits [
  if find i "n" [
    print rejoin [i "contains the letter 'n'"]
  ]
]
```

Repeat loops are often used like foreach loops, but they make use of a counter variable to pick numbered items from the series of items in the block. The 'length?' function is used to determine the number of repeats required to go through the length of the block:

```
fruits: ["apple" "orange" "banana"]
repeat i (length? fruits) [print fruits/:i]             ; prints each item
```

```
repeat i (length? fruits) [print pick fruits i]        ; the same as above
repeat i (length? fruits) [
  print rejoin [fruits "'s index number in fruits is: " fruits/:i]
]
```

'For' can be used to loop through groups (rows) of values:

```
employees: [
  "John Smith"   "1 Street Rd."     "444-2325"
  "Joe Blow"     "2 Pike Lane"      "346-2339"
  "Bill Jones"   "32 Lane St."      "344-3238"
  "Bob James"    "8 Road Pike"      "434-5820"
  "Dave Dow"     "9 Street Court"   "343-9923"
]
for i 1 (length? employees) 3 [
  print "----------------------------------------------------"
  print uppercase employees/(i)
  print rejoin [
    "Address:  " employees/(i + 1) newline "Phone:    " employees/(i + 2)]
]
```

The 'remove-each' function is used to remove any values from a block, which match a given criteria:

```
remove-each i fruits [find i "b"]                 ; removes banana
```

The 'select' function returns the next item after a specified item in a list. This has many powerful uses:

```
print select schedule "Bill"
```

The 'request-list' function is helpful in selecting items from a block of data. It returns the value which the user selects from a displayed list. As with every other function, the returned user-selected value can be assigned a word label ('name' in the example below):

```
names: ["Joe" "Bill" "Bob"]
name: request-list "Choose:" names
notify rejoin ["You chose: " name]
```

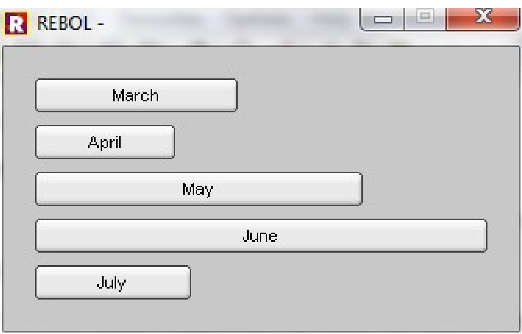
Note that the above code could be shortened by simply using the return value of the 'request-list' function as the data argument of the 'notify' function. Also, the block labeled 'names' above can be used directly as the data argument of the 'request-list' function (instead of using the 'names' label as above):

```
notify rejoin ["You chose: " request-list "Choose:" ["Joe" "Bill" "Bob"]]
```

You can generate GUI layout blocks using 'foreach' loops and other block operations, then display the block using 'view layout'. In the first line of the following example, a block of data requested from the user, is labeled 'd'. In the second line, a new empty block labeled 'g' is created. Then a foreach loop is used to go through every group of 2 month and number values (each group of 2 values is labeled 'm' and 'v' in the foreach loop), and the 'append' function is used to add button widgets with the month text, and a size of the value 'v' * 10, to the 'g' block. That generated block is then displayed with 'view layout':

```
REBOL [title: "Bar Chart Display"]
d: do request-text/default {["April" 9 "May" 21 "June" 29 "July" 10]}
```

```
g: [] foreach [m v] d [append g reduce['btn m v * 10]] view layout g
```



You can use the 'save' and 'load' functions to store and retrieve block data to/from a hard drive, thumb drive, web server, FTP server, etc:

```
names: ["Joe" "Bill" "Bob"]
save %thenames names
names: load %thenames
save ftp://user:pass@url.com/public_html/thenames
names: load http://re-bol.com/thenames
```

The 'read' function reads data byte-for-byte from file (the exact characters are loaded, one bye one as they exist sequentially on the hard drive). The 'load' function is able to perform a CONVERSION of saved data, if it recognizes common structures such as blocks, data types, images, sounds, etc. The 'write' function writes data byte-for-byte to a file. 'Write' is the storing corollary of 'read'. Data that is written will generally be 'read' back. The 'save' function can perform a CONVERSION of common data types. 'Save' is the storing corollary of 'load'. Data stored with 'save' will generally be 'load'ed back. When saving blocks and other structured data, the 'save' and 'load' functions provide a simple way of storing and retrieving those complex memory models. 'Read' and 'write' provide more fine grained control of how the information is handled, but can take some more work on your part to ensure data is properly formatted during storage and retrieval:

```
write %thenames.txt mold names       ; write + 'mold' saves a block
write/append %thenames.txt "Tim"     ; append data values directly to file
names: load %thenames.txt            ; 'load' correctly loads that file
names: to-block read %thenames.txt   ; 'to-block' loads the read data

write/lines %file.csv ["line1" "line2" "line3"] ; writes each item to line
csv: read/lines %file.csv   ; load file as a block, with each line as item
```

It's important to understand that the examples shown in this section all demonstrate simple syntax patterns using simple values, but the data values stored in blocks can be complex information of *any* type: images, sounds, videos, other binary file types, etc.:

```
myimages: copy []
foreach img [logo.gif stop.gif exclamation.gif] [
  append myimages do img
]
foreach img myimages [probe img  print type? img]
```

Learning to use and manipulate blocks of data (also called 'series') is one of the most important steps in mastering Rebol. You'll use them in dealing with every sort of data encountered in computing (not just simple lists of values, but also text strings, and things like graphics coordinates that make games operate, databases of information which make CRUD apps operate, etc. You'll also find that you can achieve more fine grained control of emails in email accounts, files in a file system, streams of data in a network socket which make network apps operate, etc., all using the exact same series functions, upon 'ports', which give

access to the most common data sources used in all varieties of apps.

5. Flow Control (conditional evaluations and loops)

Use the following code structure to handle conditional evaluations: if (this is true) [do this]

```
if (now/time = 12:00) [notify "It's noon."]
if now/time = 12:00 [notify "It's noon."]          ; parentheses are optional
if now/time > 12:00 [notify "It's after noon."]   ; greater than
if now/time < 12:00 [notify "It's morning."]      ; less than
if now/time <> 12:00 [notify "It's not noon."]    ; not equal to
```

'Either' does one block of actions if the evaluation is true, another if false:

```
either now/time > 8:00am [
    notify "It's time to get up!"
][
    notify "You can keep on sleeping."
]
```

'Case' can be used to choose between a variety of actions, depending on the situation:

```
name: "john"
case [
    find name "a" [alert {Your name contains the letter "a"}]
    find name "e" [alert {Your name contains the letter "e"}]
    find name "i" [alert {Your name contains the letter "i"}]
    find name "o" [alert {Your name contains the letter "o"}]
    find name "u" [alert {Your name contains the letter "u"}]
    true [alert {Your name doesn't contain any vowels!}]
]
```

The 'all' structure here checks if evaluation1 AND evaluation2 AND evaluation3 are all true:

```
if all [6:00pm > 12:00pm   6:00am < 12:00pm   9:00pm <> 12:00pm] [
    print "all those evaluations are true"
]
```

The 'any' structure here checks if any one of evaluation1 OR evaluation2 OR evaluation3 is true:

```
if any [6:00pm = 12:00pm   6:00am = 12:00pm   9:00pm < 12:00pm] [
    print "any 1 of those evaluations is true"
]
```

'Forever' loops continue until the program ends, or until the 'break' function is evaluated:

```
forever [
  print "Wait till 6pm, or press [ESC] to break"
  if now/time = 6:00pm [notify "It's 6pm"  BREAK]
]
```

'While' loops continue while the given evaluation is true:

```
while [now/date < 21-dec-2202] [
  print "Press [ESC] to break"
]
```

'Repeat' loops increment a counter variable a given number of times:

```
repeat count 50 [print rejoin ["This is loop #: " count]]
repeat i 50 [print rejoin ["This is loop #: " i]]
```

'For' loops can handle more complex counting patterns ('repeat' loops perform faster, and so they're more commonly used in Rebol):

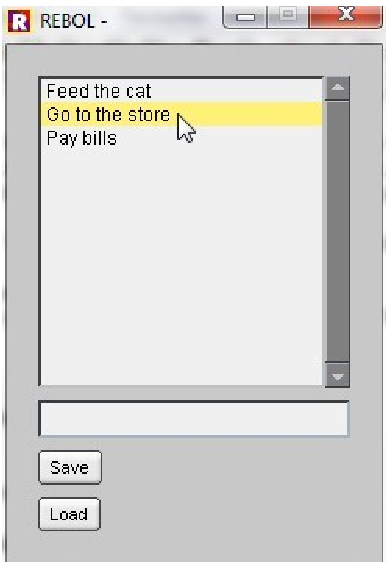
```
for counter 10 1 -2 [print counter]
for day 1-may-2017 1-dec-2017 14 [print day]
for dimes $15.00 $16.50 $.10 [print dimes]
```

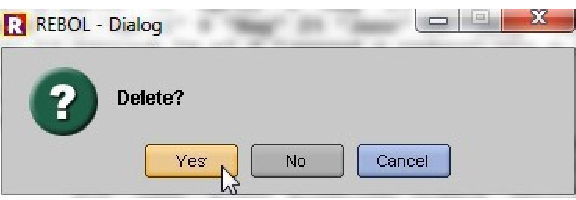
The 'attempt' and 'if error? try' structures can be used to handle errors:

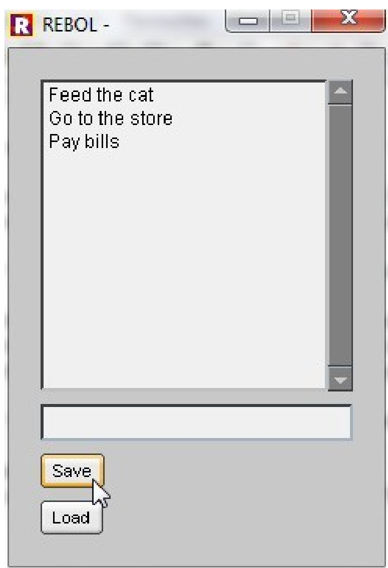
```
attempt [0 / 0]                                    ; ignore error
if error? try [0 / 0] [notify "Divide by 0 error"]   ; if error [do this]
```

6. More Example Apps

6.1 To-Do List





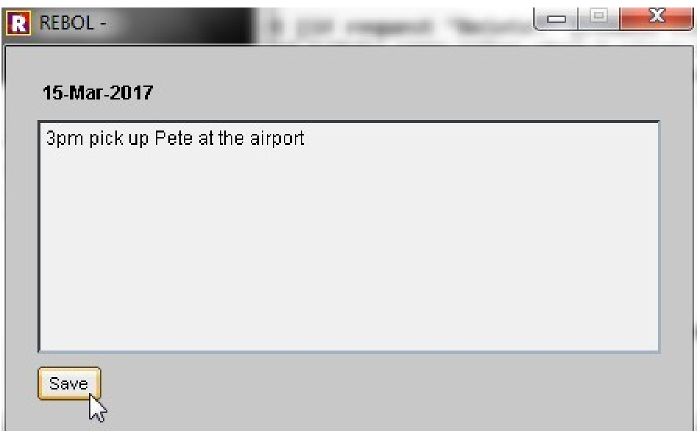


The example allows the user to add, save, load, and click items to delete:

```
REBOL [title: "To-Do List"]
view layout [
     t: text-list [if request "Delete?" [remove find t/data value  show t]]
     field [append t/data copy value show t clear-face face]
     btn "Save" [save %todo.txt t/data  notify "Saved"]
     btn "Load" [attempt [t/data: load %todo.txt  show t]]
]
```

1. The first line is the header, with the title text which appears in the header bar of the GUI display
2. The second line begins the GUI layout.
3. A text-list widget is added to the layout. When the user clicks on the text-list:
    - A pop-up requester dialogue asks the user if the selected item in the text-list should be deleted. If the response is affirmative, then the currently selected value is removed from where it's found in the displayed data of the text-list widget, and the text-list display is updated with the 'show' function.
4. A field widget is added to the layout. When the user enters text into the field widget:
    - The entered value is appended to the data displayed in the text-list widget, and the display is updated with the 'show' function. Then the 'clear-face' function is used to erase the displayed text on the face of the field widget.
5. A button widget is added to the layout. When the user clicks the button:
    - The data values displayed in the text-list widget (labeled 't') is saved to the file %todo.txt. Then the user is notified that the operation is complete.
6. Another button widget is added to the layout. When the user clicks this button:
    - The data in the %todo.txt file is loaded into the data block displayed by the text-list widget (labeled 't'), then the display is updated with the 'show' function. This entire operation is wrapped in an 'attempt' block, in case the user cancels the load operation (or if they choose an incorrect file to load, etc.).

6.2 Personal Scheduler



This example allows users to save, retrieve, and edit calendar events:

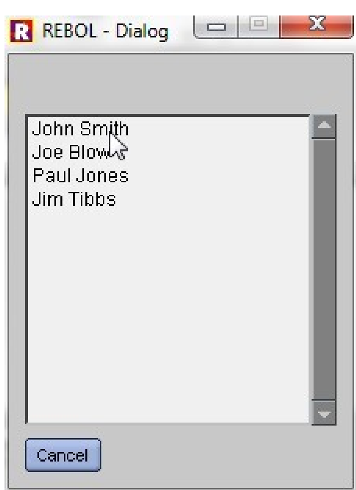
```
REBOL [title: "Personal Scheduler"]
write/append %m ""
forever [
      if not d: request-date [break]
      view center-face layout [
           text bold form d
           a: area form select (l: load %m) d
           btn "Save" [save %m head insert l reduce[d a/text] unview]
      ]
]
```

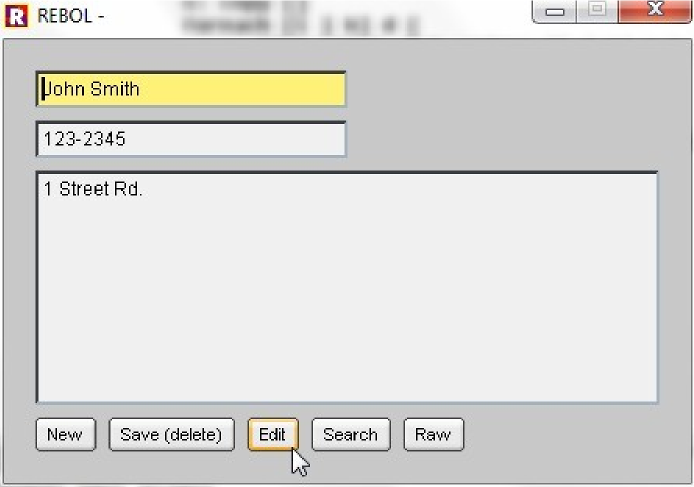
1. The first line is the header, with the title text which appears in the header bar of the GUI display
2. The second line creates an empty file named %m, if it doesn't already exist. If the file does exist, it's not affected (appending the empty string "" to a file does nothing to change it).
3. A 'forever' is begun. It runs the following actions until the 'break' function is evaluated (or until the program is otherwise ended):
    - A date requestor dialogue pops to allow the user to select a date. The selected date value is assigned the label 'd'. If the user cancels selecting a date, the 'break' function stops the forever loop, and the program ends.
    - A layout is begun. The layout is centered on screen, using the 'center-face' function.
        - A text widget is added to the layout. The text in the widget is the date value selected by the user (the 'form' function is used to convert the date value to a string of text which can be displayed by the text widget). The text in the widget is displayed using a bold font.
        - A area widget labeled 'a' is added to the layout. The text displayed in the area widget is loaded from the file %m (this loaded data is labeled 'l' here). The displayed value is selected from the loaded data - the 'select' function returns the value immediately following a found value in the data block, which in this case is the value labeled 'd' (the date chosen by the user from the pop-up requestor).
        - A button widget is added to the layout. When the button is clicked by the user:
            - A data block is created with the 'reduce' function. That block, which contains the date value labeled 'd' and the text in the area widget labeled 'a', is inserted into

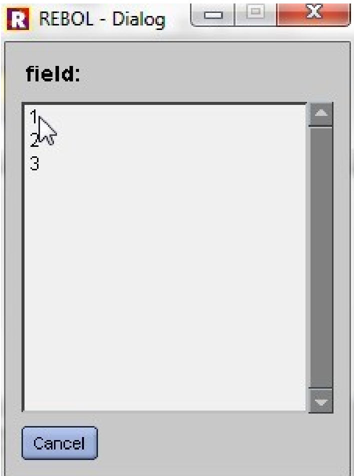
the block labeled 'l' (which contains the entire previously loaded data block from the %m file). That updated data block is saved back to the %m file. Notice that no data is deleted from the %m file during this operation. By inserting the new data into the head of the l block, the newest record associated with any given date in the block is now closest to the beginning of the block, when the 'select' function is used above to search for the text associated with a chosen date, the first value it finds is used. All the previous entries for that date are ignored. This forms an automatic edit history trail which can be used to undo changes, if desired. After the write operation is completed, the 'unview' function is used to close the layout, and the 'forever' loop continues.

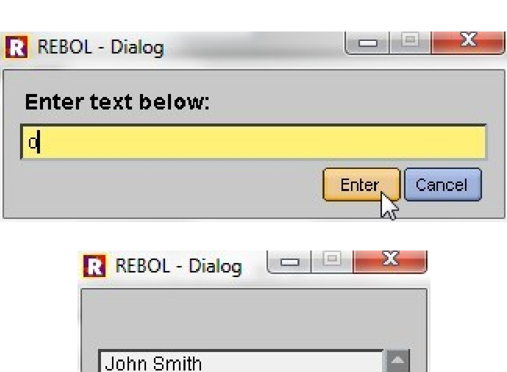
7. Generic CRUD App with Form and Data Grid Displays

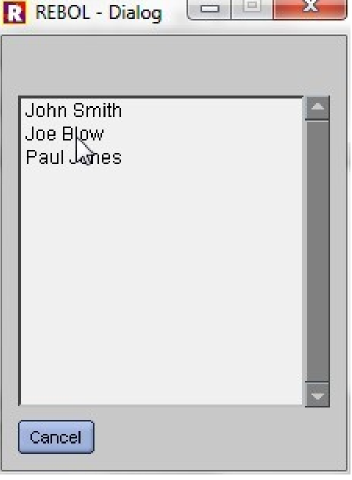
This is the most important and complex app in this tutorial. It covers many of the common techniques used to create useful data management apps:

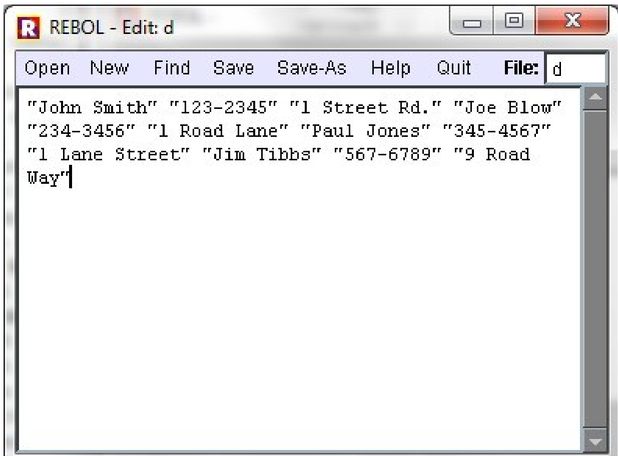

REBOL - Dialog
John Smith
Joe Blow
Paul Jones
Jim Tibbs
Cancel

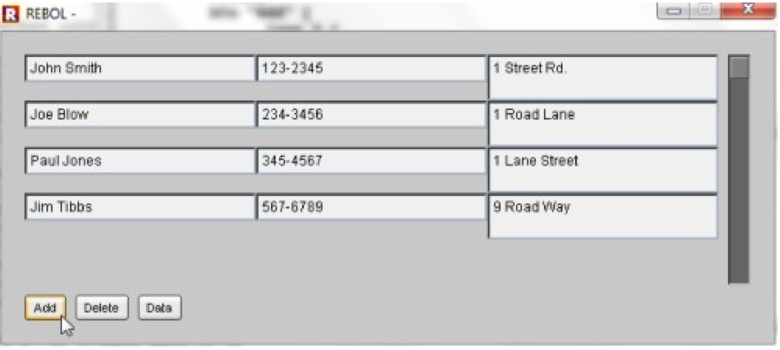

REBOL -
John Smith
123-2345
1 Street Rd.
New
Save (delete)
Edit
Search
Raw

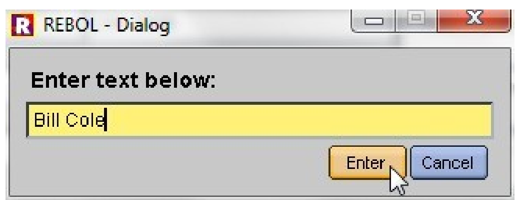

REBOL - Dialog
field:
1
2
3
Cancel

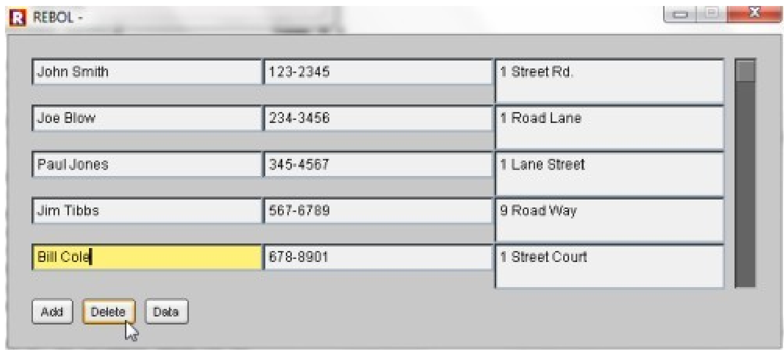

REBOL - Dialog
Enter text below:
d
Enter
Cancel

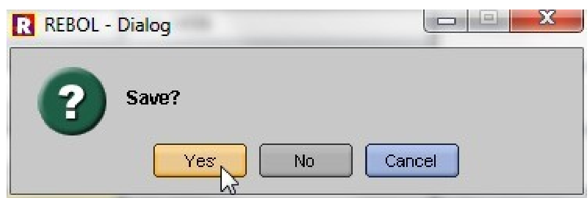

REBOL - Dialog
John Smith
Joe Blow
Paul Jones
Cancel


REBOL - Edit: d
Open New Find Save Save-As Help Quit File: d
"John Smith" "123-2345" "1 Street Rd." "Joe Blow"
"234-3456" "1 Road Lane" "Paul Jones" "345-4567"
"1 Lane Street" "Jim Tibbs" "567-6789" "9 Road
Way"


REBOL -
John Smith
123-2345
1 Street Rd.
Joe Blow
234-3456
1 Road Lane
Paul Jones
345-4567
1 Lane Street
Jim Tibbs
567-6789
9 Road Way
Add
Delete
Data







Every single bit of code is documented, line-by-line. You can reuse pieces of code in this app to produce many of the most common parts of data management apps of all sorts:

```
REBOL [title: "Generic CRUD & Data Grid Combined + Search/Add/Delete/IEF"]

; This block of code labeled 'n' works just like a function.  It's
; actions are used to clear all the fields in the screen layout:

n: [

  ; This foreach loop goes through each field in the layout (named 'a',
  ; 'b', and 'c' below), and performs the following actions:

  foreach i [a b c] [

    ; This sets the face of each widget to empty text:

    set-face get i ""

  ]

  ; 'focus' highlights and puts the cursor in the field labeled 'a':

  focus a

]

; This block of code labeled 's' saves the database (labeled 'd'
; below), to a file named 'd', then notifies the user with a pop-up
; dialogue that the operation is complete:

s: [
  save %d d
  notify "ok"
```

```
]

; This block of code labeled 'x' first runs the 'z' block, which
; deletes any pre-existing copy of the currently displayed record
; from the database, then appends the current record (the reduced
; block of text values in fields 'a', 'b', and 'c') to the database
; (labeled 'd'), then runs the 'n' block to clear the screen:

x: [
  do z
  repend d [a/text b/text c/text]
  do n
]

; This 'y' block allows the user to select from a pop-up requestor
; which lists all the first fields of each record in the database.
; Then it sets the displayed field widgets in the screen layout to
; display each of the 3 fields of data in the record:

y: [

   ; The entire operation is wrapped in an 'attempt' block, in case
   ; the user cancels the request-list operation.  Without this error
   ; handling, the 'find/skip' function would crash the app whenever
   ; the user cancels either of the requestor dialogues, because it
   ; would not have a list argument available to complete its
   ; operation:

  attempt [

     ; This lines sets the label 'r' to the result of the 'copy/part'
     ; function, which in turn takes 3 values from the result of the
     ; 'find/skip' function, which searches every 3 items in the
     ; database (labeled 'd') for the value returned by the
     ; 'request-list' function, which displays a list of every third
     ; value in the database (and that list is gotten using the
     ; 'extract' function).

     r: copy/part (find/skip d (request-list "" extract d 3) 3) 3

     ; This repeat loop goes through each field labeled 'a', 'b', and
     ; 'c' (that block of field names is labeled 'j' here), and sets
     ; the face of each field to the data item in the 'r' block
     ; (created above) which has the same index number:

     repeat i length? j: [a b c] [

       ; The face of the first field in the j block (the 'a' field) is
       ; set to display the first data value in the 'r' block (the
       ; first field of the the found data record), Then the second
       ; field ('b') is set to display the second data value, then
       ; the third field is set to the third value (etc., if there
       ; were more fields and values):

       set-face get j/:i r/:i

     ]
  ]
]

; This block of code labeled 'z' deletes a record from the database:

z: [

   ; The 'remove' loop goes through each group of 3 values in the
   ; database, temporarily labels each of those three values 'i', 'j',
   ; and 'k', during each loop, and removes any groups in which
   ; the criteria satisfies the contained evaluation:
```

```
  remove-each [i j k] d [

    ; The criteria for removing the group of 3 values is that the
    ; first of the three values ('i') match the text in the field
    ; labeled 'a':

    i = a/text

  ]
]

; When wrapping this script into an exe, we may want to switch to
; a known folder (but that's not necessary if the script is run
; from a known location, so this line is commented out unless
; needed):

; change-dir %/c/

; This line creates an empty data file (with the file name 'd'), if
; it doesn't already exist.  If the file does exist, it's not changed
; at all (the empty string is written to it, which does nothing to
; change its contents):

write/append %d ""

; The word label 'd' is assigned to the block of data loaded from the
; file named 'd':

d: load %d

; This line creates the GUI screen layout, which displays the widgets
; enclosed in the following block:

view center-face layout [

; 2 field widgets and an area widgets, labeled 'a', 'b', and 'c'
; respectively, are added to the screen layout:

  a: field
  b: field
  c: area

; By default, widgets are placed below one another in the layout.
; The 'across' word changes that default behavior, so that all
; following widgets are placed across the screen (you could shift
; back to the default behavior with the word 'below'):

  across

; This is a button widget with the text "New" displayed on its face:

  btn "New" [

    ; When the button is clicked by the user, the block of code
    ; labeled 'n' is run:

    do n

  ]

; This is a button widget with the text "Save (delete)" on its face:

  btn "Save (delete)" [

   ; When the button is clicked, the blocks of code labeled 'x' and
   ; 's' are run (see above for descriptions of what those blocks
   ; do):

    do x do s
```

```
  ][

    ; When the button is right-clicked, the blocks of code labeled
    ; 'z' and 'n' are run (see their descriptions above):

    do z do n

  ]

; This is a button widget with the text "Edit", which runs the block
; of code labeled 'y':

  btn "Edit" [
    do y
  ]

; This is a button widget with the text "Search", which runs the
; code below (this code could have been put into a named block like
; the others, but that wasn't necessary here because this code is not
; used anywhere else in the app - the other named blocks were all
; used in several places, which would have required duplicating their
; code throughout the app.  The named blocks just help avoid such
; duplication, which reduces code size and helps eliminate errors:

  btn "Search" [

; This 'if' conditional operation checks to see if either of the
; values entered by the user in the following pop-up requestors are
; empty.  If so, the function is exited (this keeps the program from
; crashing with an error that would occur when trying to process
; those empty values):

    if find reduce [

      ; The label 'p' is assigned to the value returned by the
      ; 'request-list' function (that value is a number selected by
      ; the user from the displayed list (1, 2, 3)).  This value
      ; will be used to choose the field of data to search:

      p: request-list "field:" [1 2 3]

      ; The label 'h' is assigned to the value returned by the
      ; 'request-text' function (this will be the search term to
      ; be found in the field chosen above):

      h: request-text

    ] none [

      ; The button's action block is exited if the user cancels out
      ; of either of the requestors above:

      exit

    ]

    ; A new block labeled 'u' is created to store results of the find
    ; operation below:

    u: copy []

    ; This foreach loop goes through each group of 3 items in the
    ; database (labeled 'd'):

    foreach [i j k] d [

      ; This 'if' evaluation checks whether the search term ('h') is
      ; found in the chosen field ('p') in each group of 3 values.
```

```
      ; The word 'f' is assigned to the results of the 'find'
      ; function:

      if f: find pick reduce [i j k] p h [

        ; If the find evaluation is successful, the first value of
        ; the group of 3 values (labeled 'i') is appended to the
        ; 'u' block.  This collects a list of all the header values
        ; (first value) of all the records in which the search term
        ; was found in the chosen field:

        append u i

      ]
    ]

    ; This 'either' evaluation checks whether the chosen header
    ; value (selected from the 'u' block generated above) is found
    ; in any third data value in the database (the header value in
    ; each group of 3 values in the database).  The label 'v' is
    ;  assigned to the result of that find operation:

    either v: find/skip d request-list "" u 3 [

      ; If the find evaluation is successful, then this repeat loop
      ; goes through each field 'a', 'b', 'c' (that block of fields
      ; is labeled 't' here):

      repeat i length? t: [a b c] [

        ; Each of the a, b, and c fields are set to display the 1st,
        ; 2nd, and 3rd values in the 'v' block above (the found
        ; record):

        set-face (get t/:i) v/:i

      ]
    ][

     ; If the find operation above is not successful, the user is
     ; notified:

      notify "Not found"

    ]
  ]

; This is a button widget with the text "Raw" on its face:

  btn "Raw" [

    ; When the button is clicked, the raw database file is opened
    ; with Rebol's built in text editor (be careful if you edit -
    ; if any items are deleted or add the database format would be
    ; corrupted because the foreach loops would no longer pick out
    ; correct field values for each record):

    editor %d

  ]
]

; The first part of the app is complete (the GUI layout block is
; closed on the line above).  Now we begin to add some action
; blocks which will be used by the data grid app:

; This block of actions labeled 'lo' is run immediately:

do lo: [
```

```
; This foreach loop goes through each group of 3 items in the
; database:

  foreach [i j k] d [

    ; We begin generating a block of layout code labeled 'o'.
    ; In this block, widgets are laid out across the screen, with
    ; no empty space in between:

    append o: [origin 0x0 across space 0x0] reduce [

      ; During each iteration of the foreach loop (which takes a
      ; group of every 3 items in the database), 2 field widgets
      ; and a small area widget are added, each displaying one of
      ; the group of 3 values from the database.  A 'return word
      ; is then used to place the next items in the layout on a
      ; new line:

      'field i
      'field j
      'area 200x40 k
      'return

    ]
  ]

  ; If the 'o' word has no value, that means that nothing was added
  ; to the layout in the loop above (because the database was empty).
  ; If this is the case, the program is ended with the q (quit)
  ; function:

  if not value? 'o [q]

  ; The 'layout' function is used below to generate a screen layout
  ; from the 'o' block above.  The screen layout is assigned the
  ; label 'p':

  p: layout o

]

; This 'ss' function updates the vertical (y) scroll position of the
; 'p' layout above (which is placed in the pane of the box labeled
; 'g' below):

ss: does [

  ; The update calculation is based upon the difference in vertical
  ; size between the pane which displays the 'p' layout and the
  ; vertical size of the box labeled 'g'.  This difference is
  ; multiplied by the position of the slider widget labeled 'sd'
  ; below:

  g/pane/offset/y: g/size/y - g/pane/size/y * sd/data

  ; The 'show' function is used to update the display of any the 'g'
  ; widget ('show' is used whenever a widget's face display is
  ; changed):

  show g

]

; This 'sv' function collects the data values contained in every
; widget displayed in the 'p' layout (which is displayed on a pane
; in the box labeled 'g'):

sv: does [
```

```
    ; This foreach loop goes through every widget in the layout:

    foreach i g/pane/pane [

      ; And each one of those values is appended consecutively to a
      ; block labeled 'z':

      append z: [] i/text

    ]

    ; Then that block is saved to the file labeled 'd'.  This whole
    ; operation has the effect of saving the current database values
    ; displayed on screen (along with any edits made to that data),
    ; to the file:

    save %d z

  ]

  ; This 'insert-event-function' adds an event handler to the program:

  insert-event-func [

    ; When a 'close' event occurs (when the user closes the program
    ; window):

    e: :event
    either e/type = 'close [

      ; This 'if' condition asks the user if the data should be saved.
      ; If the user responds yes, then the 'sv' function is run, to
      ; save the data to a file:

      if request "Save?" [sv]

      ; Then the event labeled 'e' is run (the program is closed):

      e

    ][

      ; Events other than 'close' events are simply evaluated (passed
      ; along):

      e

    ]
  ]

  ; This function performs a screen update.  It's used whenever the
  ; data in the database is altered:

  u: does [

    ; First, the 'lo' function is run, then the layout displayed in
    ; the pane of the box labeled 'g' is updated to the current value
    ; represented by 'p', the position of the layout is reset, and the
    ; 'ss' function is run to sync that position to the position of
    ; the slider widget:

    do lo
    g/pane: p
    g/pane/offset: 0x0
    ss

  ]
```

```
; Now a new screen layout is created and displayed:

view center-face layout [

  ; This 'across' word forces all following widgets to be placed
  ; horizontally across the screen:

  across

  ; A box widget labeled 'g' is added to the layout.  The size of
  ; this widget is a coordinate pair equal to the horizontal size of
  ; the 'p' layout across, by a vertical size of 200 pixels down:

  g: box (as-pair p/size/x 200) with [

    ; The pane of the box widget is set to display the 'p' layout
    ; and the initial scroll position is set:

    pane: p
    pane/offset: 0x0

  ]

  ; A slider widget labeled 'sd' is added (20 pixels wide by 200
  ; pixels tall):

  sd: slider 20x200 [

    ; Whenever the user moves the slider, the 'ss' function is run
    ; (see the description of the 'ss' function above):

    ss

  ]

  ; This 'return' word starts a new line in the layout (similar to
  ; a carriage return, but instead of characters, an new line of
  ; widgets is begun):

  return

  ; This is a button widget with the text "Add" on its face:

  btn "Add" [

    ; When the user clicks the button, the following actions are
    ; performed 3 times:

    loop 3 [

      ; The database (labeled 'd') is appended with whatever the
      ; user types into the 3 pop-up text requestor dialogues:

      append d request-text

    ]

    ; The 'u' function is run to update the display:

    u

  ]

; This is a button widget with the text "Delete" on its face:

  btn "Delete" [

    ; When the user clicks the delete button, the following
    ; remove/part function removes 3 values (1 record) from the
```

```
        ; position in the database where the item requested from the
        ; user is found.  The item requested from the user is chosen
        ; from a list of  every three items in the database (the
        ; record 'header' values):

        remove/part find/skip d (request-list "" extract d 3) 3 3

        ; After the record is removed, the layout display is updated
        ; with the 'u' function:

        u

    ]

; A button widget with the text "Data" is added to the layout:

    btn "Data" [

        ; When the user clicks this 'data' button, the following
        ; 'foreach' loop goes through each widget in the grid layout:

        foreach i g/pane/pane [

            ; The text of each widget in the layout is appended to the
            ; block labeled 'z' (so the 'z' block includes any edits
            ; which the user may have made to the displayed data):

            append z: [] i/text

        ]

        ; Rebol's built-in text editor is used to display the 'z' block
        ; to the user:

        editor z

    ]
]
```

Here's the entire program, without any comments. Try adjusting parts of the code to make the app perform differently. Add data fields, change the size and layout of widgets in the user interface, change word labels and file names, alter the way the search works, etc. Play with the code, break it, and try to figure out how to fix it:

```
REBOL [title: "Generic CRUD & Data Grid Combined + Search/Add/Delete/IEF"]
n: [
    foreach i [a b c] [
        set-face get i ""
    ]
    focus a
]
s: [
    save %d d
    notify "ok"
]
x: [
    do z
    repend d [a/text b/text c/text]
    do n
]
y: [
    attempt [
        r: copy/part (find/skip d (request-list "" extract d 3) 3) 3
        repeat i length? j: [a b c] [
            set-face get j/:i r/:i
        ]
    ]
```

```
]
z: [
  remove-each [i j k] d [
    i = a/text
  ]
]
; change-dir %/c/
write/append %d ""
d: load %d
view center-face layout [
  a: field
  b: field
  c: area
  across
  btn "New" [
    do n
  ]
  btn "Save (delete)" [
    do x do s
  ][
    do z do n
  ]
  btn "Edit" [
    do y
  ]
  btn "Search" [
    if find reduce [
      p: request-list "field:" [1 2 3]
      h: request-text
    ] none [
      exit
    ]
    u: copy []
    foreach [i j k] d [
      if f: find pick reduce [i j k] p h [
        append u i
      ]
    ]
    either v: find/skip d request-list "" u 3 [
      repeat i length? t: [a b c] [
        set-face (get t/:i) v/:i
      ]
    ][
      notify "Not found"
    ]
  ]
  btn "Raw" [
    editor %d
  ]
]
do lo: [
  foreach [a b c] d [
    append o: [origin 0x0 across space 0x0] reduce [
      'field a
      'field b
      'area 200x40 c
      'return
    ]
  ]
  if not value? 'o [q]
  p: layout o
]
ss: does [
  g/pane/offset/y: g/size/y - g/pane/size/y * sd/data
  show g
]
sv: does [
  foreach i g/pane/pane [
    append z: [] i/text
```

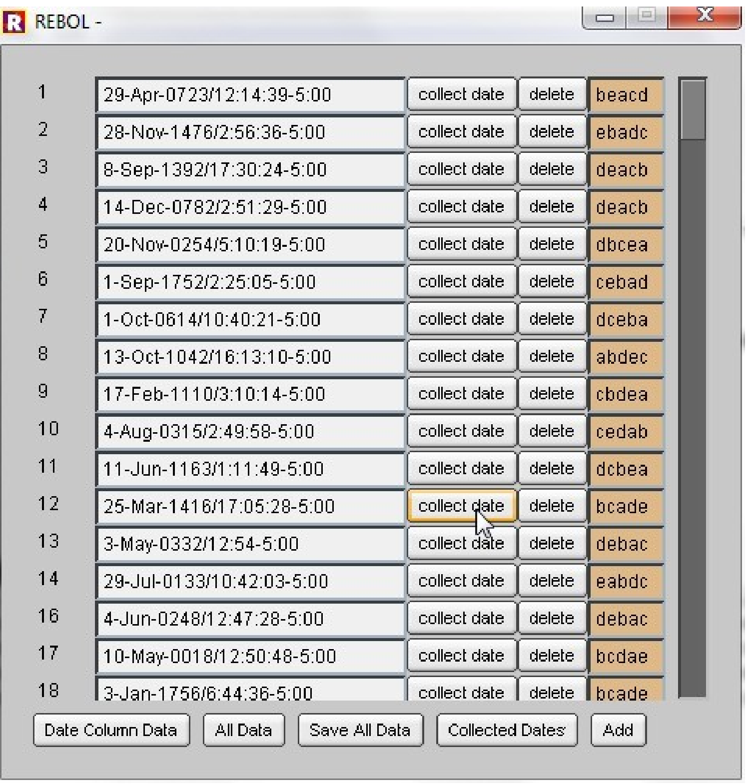
```
    ]
    save %d z
]
insert-event-func [
    e: :event either e/type = 'close [
        if request "Save?" [sv]
        e
    ][
        e
    ]
]
u: does [
    do lo
    g/pane: p
    g/pane/offset: 0x0
    ss
]
view center-face layout [
    across
    g: box (as-pair p/size/x 200) with [
        pane: p
        pane/offset: 0x0
    ]
    sd: slider 20x200 [
        ss
    ]
    return
    btn "Add" [
        loop 3 [
            append d request-text
        ]
        u
    ]
    btn "Delete" [
        remove/part find/skip d (request-list "" extract d 3) 3 3
        u
    ]
    btn "Data" [
        foreach i g/pane/pane [
            append z: [] i/text
        ]
        editor z
    ]
]
```

And here's the exact same program again, packed into a very dense ~21 lines of code:

```
REBOL [title: "Generic CRUD & Data Grid Combined + Search/Add/Delete/IEF"]
n: [foreach i[a b c][set-face get i""]focus a]  s: [save %d d  notify"ok"]
x: [do z repend d[a/text b/text c/text]do n]  ief: :insert-event-func
y: [attempt[r: copy/part(find/skip d (request-list"" extract d 3)3)3
  repeat i length? j:[a b c][set-face get j/:i r/:i]]]  ; change-dir %/c/
z: [remove-each[i j k]d[i = a/text]]   write/append %d ""  d: load %d
view center-face layout[a: field b: field c: area across btn "New"[do n]
  btn "Save (delete)"[do x do s][do z do n] btn "Edit"[do y] btn "Search"[
    if find reduce[p: request-list"field:"[1 2 3]h: request-text]none[exit]
    u: copy[] foreach[i j k]d[if f: find pick reduce[i j k]p h[append u i]]
    either v: find/skip d request-list"" u 3 [repeat i length? t:[a b c][
      set-face (get t/:i) v/:i]][notify "Not found"]] btn "Raw"[editor %d]]
do lo: [foreach [a b c] d [append o: [origin 0x0 across space 0x0] reduce [
  'field a 'field b 'area 200x40 c 'return]]if not value? 'o[q]p: layout o]
ss: does [g/pane/offset/y: g/size/y - g/pane/size/y * sd/data show g]
sv: does [foreach i g/pane/pane [append z: [] i/text] save %d z]
ief [e: :event either e/type = 'close [if request "Save?" [sv] e][e]]
u: does [do lo  g/pane: p g/pane/offset: 0x0 ss]  view center-face layout [
  across  g: box (as-pair p/size/x 200) with [pane: p  pane/offset: 0x0]
  sd: slider 20x200 [ss]  return btn "Add"[loop 3[append d request-text] u]
```

```
  btn "Delete"[remove/part find/skip d(request-list"" extract d 3)3 3 u]
  btn "Data"[foreach i g/pane/pane [append z:[] i/text] editor z]
]
```

Here's a more feature packed version of the data grid portion of app above. You should be able to follow the code, if you understand the previous example:




```
REBOL [title: "More Complex Data Grid"]
if not exists? %griddata [
  repeat i 299 [
    append mydata: [] reduce [
      i
      random now
      random "abcde"
    ]
  ]
  save %griddata mydata
]
loaded-data: load %griddata
dlt: func [cnt] [
  if true = request "Really?" [
    remove/part at loaded-data cnt 3
    do lo
    g/pane: p
    g/pane/offset: 0x0
    settoslider
  ]
]
collect-all: does [
  foreach i g/pane/pane [
    if not find ["collect date" "delete"] i/text [
      append y: [] i/text
```

```
    ]
  ]
]
settoslider: does [
  g/pane/offset/y: g/size/y - g/pane/size/y * slid/data
  show g
]
saved: copy []
do lo: [
  count: 1
  foreach [a b c] loaded-data [
    append grid: [origin 0x0 across space 0x0] reduce [
      'text 40 form a
      'field form b
      'btn "collect date" compose [append saved (form b)]
      'btn "delete" compose [dlt (count)]
      'info tan 50 form c [alert face/text]
      'return
    ]
    count: count + 3
  ]
  p: layout grid
]
insert-event-func [
  either event/type = 'close [
      collect-all
      save %griddata y
      request/ok/timeout "Database saved" .7
      event
  ][event]
]
view center-face layout [
  across
  g: box (as-pair p/size/x 400) with [pane: p  pane/offset: 0x0]
  slid: slider 20x400 [settoslider]
  return btn "Date Column Data" [
    foreach [a b c d e] g/pane/pane [
      append z: [] b/text
    ]
    editor z
  ]
  btn "All Data" [
    collect-all
    editor y
  ]
  btn "Save All Data" [
    collect-all
    save request-file/only/save/file %griddata y
  ]
  btn "Collected Dates" [editor saved]
  btn "Add" [
    append loaded-data reduce [
      1 + do pick loaded-data ((length? loaded-data) - 2)
      request-text
      random "abcde"
    ]
    do lo
    g/pane: p
    g/pane/offset: 0x0
    settoslider
  ]
]
```

8. Grids with the 'List' Widget

8.1 Display Tens of Millions of Lines, With Fast Performance

The generated grid screens above can be used to display small blocks of data, in layouts which are as versatile as you can imagine implementing. But such generated layouts will perform slowly when displaying large blocks of data. The built-in 'list' widget should be used to display large blocks of data (thousands to tens of millions of values). The following example demonstrates the default syntax needed to display data in the 'list' widget. Notice the 's' variable, the 'supply' code block, and the slider widget code block. These sections of code are mostly boilerplate - you should simply memorize their syntax and learn how to use them by examining more example code:

```
REBOL [title: "1 MILLION ROWS Grid"]

; This first line sets some labels and creates a random block of data
; labeled 'x', which will be displayed in the grid.  The value labeled
; 's' is used to store the position of the slider (used to position the
; scroll of the grid display).  The label 'asc' is used to store a
; true/false value that determines whether sort operations are performed
; ascending or descending.  The repeat loop runs 1 million times,
; incrementing the 'i' counter.  During this loop the 'x' block is
; appended with these reduced values:  the 'i' counter value, a random
; date-time value, and a random string of 4 characters (a random order of
; the letters "abcd").  Note that the 'repend' function is just a
; shortcut for the 'append' and 'reduce' functions together.  The /only
; refinement of the 'repend' function appends those values as a sub-block
; of the 'x' block:

s: 0  repeat i 1000000 [repend/only x:[] [i random now random "abcd"]]

; Here, a GUI screen layout is started:

view layout [

; The 'across' word is used to place each new consecutive widget across
; the screen.
; A 'list' (grid) widget 310x410 pixels, labeled 'l', is added to the
; screen layout.  Each row in the grid will be made up of a tan 'text'
; widget 55 pixels across, and 2 default 'text' widgets (170 and 80
; pixels), placed across the screen.  The 'supply' line is basically
; a stock line that you can memorize to use in most standard 'list'
; widget displays.  This line uses the preset 'count' word to refer to
; each consecutive row of the table data, and the preset 'index' word
; to refer to each consecutive column item within each of those rows.
; In this stock line of code, the list's row layout (defined in the
; first line below) is aligned with the 's' value, which is adjusted
; by the slider widget (the next widget in the screen layout).  The
; text on the face of each widget in the row is set to the values
; picked using those 'count' and 'index' values (or none when the end
; of the 'x' data is reached):

  across l: list 310x400 [across text 55 tan text 170 text 80] supply [
    face/text: either e: pick x count: count + s [e/:index][none]

; A 'slider' widget 20 pixels tall by 400 pixels across is added to the
; screen layout.  When the slider is moved by the user, the 's' value is
; set to the value represented by the slider position, times the length
; of the data block, labeled 's' (the 'length' function returns the
; number of rows in the 'x' data block).  The list widget labeled 'l'
; with the 'show' function, every time the slider is moved.  Then the
; 'return' word is added to start a new line in the screen layout:

  ] sl: slider 20x400 [s: value * length? x show l] return
]
```

Here's the entire script, without any comments. It's only 5 lines of code (and one of those lines is used to generate the displayed random grid data, while another is simply the standard 'view layout' needed to create any screen layout):

REBOL -

1	29-Apr-0723/4:48:23-5:00	cbad
2	14-Nov-1919/3:52:18-5:00	cabd
3	25-Dec-1260/8:04:18-5:00	abcd
4	30-May-0295/12:15:19-5:00	dabc
5	18-Feb-0738/1:57:20-5:00	acdb
6	22-Jan-0819/14:46:22-5:00	bacd
7	15-Jun-0973/1:56:15-5:00	dbca
8	21-Nov-1648/5:14:45-5:00	cdab
9	13-Nov-1609/12:12:17-5:00	acdb
10	17-Feb-1110/15:34:16-5:00	bcda
11	19-Jul-0731/12:17:53-5:00	adbc
12	1-Apr-0906/11:59:06-5:00	dacb
13	15-Aug-1621/2:52:44-5:00	cbad
14	16-Jul-1979/5:26:55-5:00	dcab
15	7-Aug-1646/11:55:21-5:00	cdab
16	27-Jan-1773/13:12:36-5:00	cbda
17	10-Jan-1613/6:43:43-5:00	bdac
18	31-Dec-0309/1:08:34-5:00	dacb
19	10-May-0018/10:25:12-5:00	dbac
20	14-Oct-1747/0:28:18-5:00	dabc
21	22-Jun-0483/11:32:51-5:00	dabc

```
REBOL [title: "1 MILLION ROWS Grid"]
s: 0  repeat i 1000000 [repend/only x:[] [i random now random "abcd"]]
view layout [
  across l: list 310x400 [across text 55 tan text 170 text 80] supply [
    face/text: either e: pick x count: count + s [e/:index][none]
  ] sl: slider 20x400 [s: value * length? x  show l]
]
```

8.2 A Practical Set of Features

Here's a more complex grid example which demonstrates how to add a number of commonly needed features: add a new row of data, remove a row, edit existing fields, save and load grid data, search, sort by column, and arrow key scroll. Notice the 'edit' and 'srt' functions (you can label them whatever you want), as well as the 'style' and 'key keycode' widget code. Memorize their syntax (or just copy/paste right now), to begin putting them to use:

```
REBOL [title: "Grid w/ Features"]; edit, find, sort, save/load, key scroll

; This first line sets some labels and creates a random block of data
; labeled 'x', which will be displayed in the grid (see the previous
; example for a complete explanation):

s: 0  asc: 1  repeat i 999 [repend/only x:[] [i random now random "abc"]]

; This 'edit' function will be used to update the faces of widgets with
; some text requested from the user (using a request-text pop-up
; dialogue).  The default text shown in the pop-up requestor is the
; text currently displayed on the face of the widget.  When the update
; operation is complete, the display of the list widget (the grid),
; labeled 'l', is updated using the 'show' function:

edit: func [f] [f/text: request-text/default f/text show l]
```

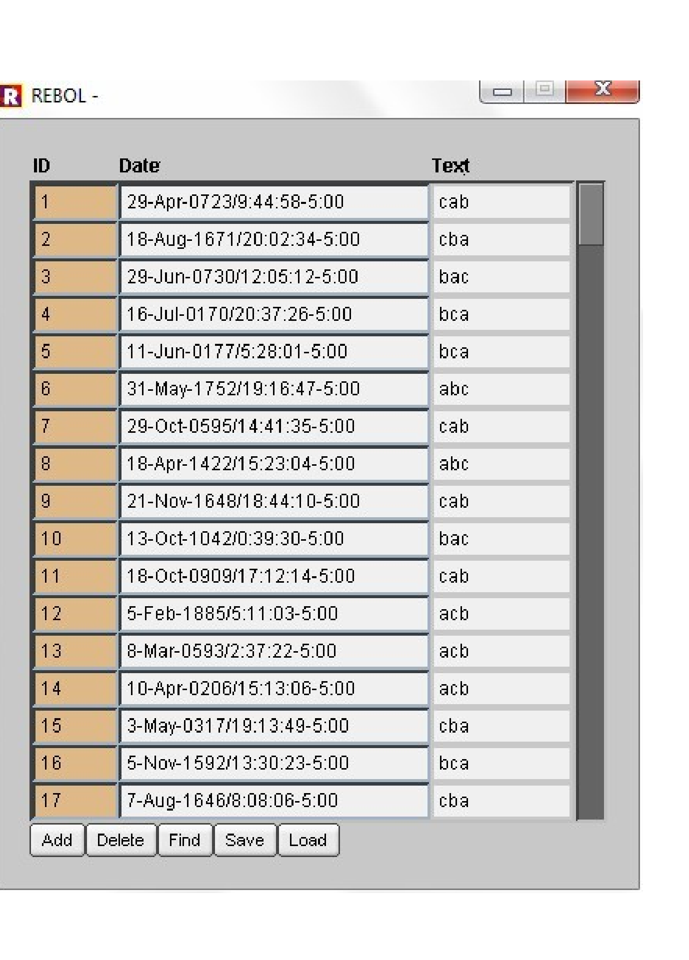
```
; This 'srt' function will be used to sort the grid data by user-selected
; columns.  The 'c' argument holds the number of the column to be sorted.
; The 'sort/compare' function takes the 'x' block (the data which fills
; the entire grid), and another function which performs the type of
; comparison to use for the sort operation (in this case, using the
; column represented by the 'c' argument, and performing a less-than
; comparison).  The 'not' function is applied to the 'asc' value
; (originally set to 1, or TRUE, above), so that every time the sort
; function is run, the sort is alternately left as-is, the row order of
; the entire 'x' block of data labeled reversed (if 'asc' is true, the
; 'reverse' function is applied to 'x').  When the sort operation is
; complete, the display of the list widget, labeled 'l', is updated
; using the 'show' function:

srt: func [c] [sort/compare x func [a b] [(at a c)< at b c]
  if asc: not asc [reverse x] show l]

; Here, a GUI screen layout is started:

view g: layout [

; A new widget style labeled 't' is created, based upon the existing
; 'text' style, 91 pixels horizontal by 24 pixel vertical, with an edge
; size of 2x2 pixels.  When any 't' style is clicked, the 'edit' function
; above is run upon its face.  The 'across' and 'space 0x0' directives
; are used place all following widgets consecutively across the screen,
; 0x0 pixels apart from one another:

  style t text 91x24 edge [size: 2x2][edit face]  across  space 0x0

; Three 'h4' (bold text) widgets of different sizes are added to the
; screen layout.  They're meant to be descriptive headers for each of
; the columns in the grid.  When any of these headers is clicked, a sort
; operation is performed on a specified column of the grid data.  The
; 'return' word is used to start a new line in the display:

  h4 "ID" 55 [srt 1]  h4 "Date" 200 [srt 2]  h4 "Text" [srt 3]  return

; A 'list' (grid) widget 350x412 pixels, labeled 'l', is added to the
; screen layout.  Each row in the grid will be made up of a tan 'info'
; widget 55 pixels across, a default 'info' widget (200 pixels across),
; and one of the 't' widgets (defined above), placed across the screen,
; with 0x0 pixels between each widget.  The 'supply' line is basically
; a stock line, as described in the previous example:

  l: list 350x412 [across  space 0x0  info 55 tan  info  t] supply [
    face/text: either e: pick x count: count + s [e/:index] [none]

; A 'slider' widget 20 pixels tall by 412 pixels across is added to the
; screen layout, as explained in the previous example:

  ] sl: slider 20x412 [s: value * length? x show l] return

; A button widget is added to the layout, displaying the text "Add" on
; its face.  When this button is clicked by the user, the 'maximum-of'
; function is used to determine the highest key value in each row (the
; first column in each row of the 'x' data block is unique integer
; number).  The return value of the 'maximum-of' function is assigned
; the label 'y'.  The 'repend' function is used, just as in the first
; line of the program, to add a new row of data to the 'x' block.  The
; first value in the row is 1 + the first value in the highest numbered
; key row (basically, this ensures that every new row gets a unique key
; value that is one above the existing highest key number in the data
; block, no matter how the grid is sorted or displayed).  The second
; value in the added row is a random date value, and the thirst value
; in the row is an empty string of text.  When the add operation is
; complete, the 'show' is used to update the display of the list widget
; (labeled 'l'):
```

```
  btn "Add"[y: maximum-of x repend/only x[1 + y/1/1 random now cp""]show l]

; A button widget is added to the layout, displaying the text "Delete".
; When this button is clicked by the user, the 'request-text' function
; pops up a dialogue which asks the user to input a number.  The return
; value of that function, converted from text to a number value using the
; 'do' function, is assigned the label 'r'.  Then the 'remove-each'
; function is used to go through the 'x' block (using the word 'i' to
; refer to each successive row).  Any row in which the first element is
; equal to the value represented by 'r', is removed from the 'x' block.
; Then the 'show' function is used to update the display of the 'l'
; list widget:

  btn "Delete" [r: do request-text/title"#" remove-each i x[i/1 = r]show l]

; A button widget is added to the layout, displaying the text "Find".
; When this button is clicked by the user, the value returned by the
; 'request-text' function (whatever the user types into the pop-up
; requestor it displays) is assigned the label 'h'.  A new block labeled
; 'u' is then created (the 'cp' is just a built-in shortcut for the
; 'copy' function).  Next, a series of 'foreach' loops goes through each
; row of data in the 'x' block (each row is labeled 'i' in the first
; foreach loop), and then each value in each row (labeled 'j' in the
; inner 'foreach' loop), and the 'find' function is used to check whether
; the search term entered by the user is found in any of those value.
; If the 'find' function returns a value (other than none), then a
; 'form'ed copy of the row data, labeled 'i' is added to the 'u' block.
; (the formed value is a single string of text, rather than a block of
; separate values).  The 'request-list' function is then used to display
; a pop-up selector that allows the user to select from the rows in
; which the search term was found (the values stored in the 'u' block).
; If the user cancels that selection, or if there are no found values,
; then the entire search operation is existed.  The first items in the
; row selected by the user is the separated from the formed row text
; (using the 'parse' function) and that value is converted to a number
; (using the 'do' function).  That value, minus 1, is assigned to the
; 's' variable, which is used by the slider widget to set the scroll
; position of the grid display.  This has the effect of scrolling
; the grid display to the row selected by the user.  The 'show' function
; is used to update the list display:

  btn "Find" [h: request-text  u: cp []
    foreach i x [foreach j i [if find form j h [append u form i]]]
    if not f: request-list"" u [exit]  s: (do first parse f "")- 1 show l]

; A button widget is added to the layout, displaying the text "Save".
; When this button is clicked by the user, the 'save' function is run.
; The file argument of the 'save' function is selected by the user, using
; the 'request-file' function (the /only refinement limits the user's
; choice to one file, the /save refinement display a save (rather than an
; open) version of the dialogue, and the /file refinement allows us to
; suggest a default file name in the pop-up dialogue).  The value saved
; to the selected file is the block of data labeled 'x'.  THIS BLOCK OF
; DATA INCLUDES ANY CHANGES MADE BY THE USER IN THE DISPLAYED GRID
; LAYOUT.  The list widget automatically handles adjusting the values in
; it's displayed block of data (labeled 'x' here), whenever the user
; edits, adds, or removes any values in any rows.  The entire save
; operation is wrapped in an 'attempt' block, in case the user cancels
; or otherwise causes the file selection process to produce an error:

  btn "Save" [attempt [save request-file/only/save/file %x.txt x]]

; A button widget is added to the layout, displaying the text "Load".
; When this button is clicked by the user, the 'load' function is run.
; The file argument of the 'load' function is selected by the user, using
; the 'request-file' function (the /only refinement limits the user's
; choice to one file, and the /file refinement allows us to suggest a
; default file name in the pop-up dialogue).  The block of data value
; loaded from the selected file is assigned the label 'x'.  The list
```

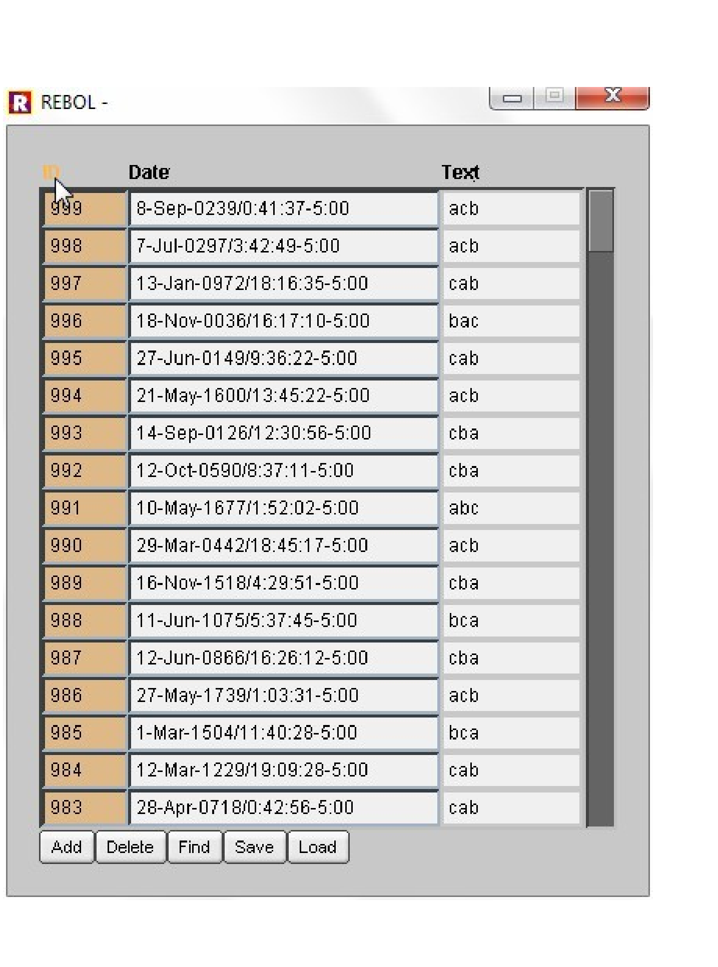
```
; widget automatically handles adjusting the values in it's displayed
; area, to the new block of data assigned to it (labeled 'x' here).
; The entire load operation is wrapped in an 'attempt' block, in case
; the user cancels or otherwise causes the file selection process to
; produce an error:

  btn "Load" [attempt [x: load request-file/only/file %x.txt show l]]

; An invisible key handler is added to the screen layout.  When the
; down arrow key on the keyboard is pressed by the user, the data value
; of the slider widget, labeled 'sl', is set to the current 's' value
; plus 10, times the number of rows in the 'x' block.  This has the
; effect of scrolling the grid display down 10 rows.  This option isn't
; really needed in this example, but it is very important in grids which
; display many thousands to millions of rows of data (the slider is not
; fine grained enough to scroll through every single row, and the key
; scroll allows you to zero on precisely desired rows).  You could
; add an up key scroll feature by subtracting 10 from the 's' value, or,
; for example a page-up/page-down option which leaps through the data
; in larger increments:

  key keycode [down] [sl/data: (s: s + 10)/ length? x show g]

]
```

Here's the entire script, without any comments. It's actually only 17 lines of (dense, but readable) code.

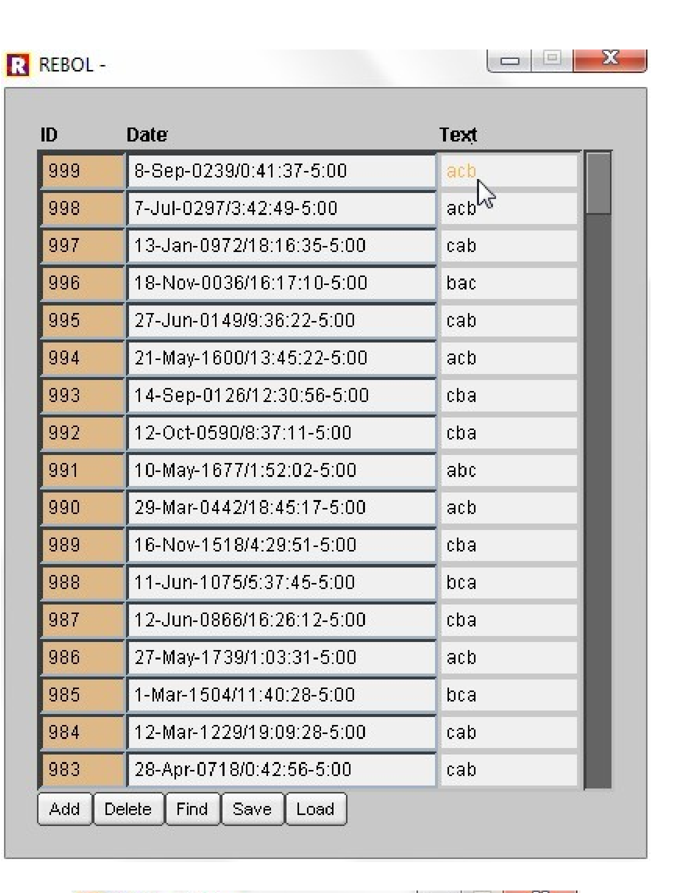
```
REBOL [title: "Grid"]; edit, add, delete, find, sort, save/load, key scroll
s: 0  asc: 1  repeat i 999 [repend/only x:[] [i random now random "abc"]]
edit: func [f] [f/text: request-text/default f/text show l]
srt: func [c] [sort/compare x func [a b] [(at a c)< at b c]
  if asc: not asc [reverse x] show l]
view g: layout [
  style t text 91x24 edge [size: 2x2][edit face]  across  space 0x0
  h4 "ID" 55 [srt 1]  h4 "Date" 200 [srt 2]  h4 "Text" [srt 3]  return
  l: list 350x412 [across  space 0x0  info 55 tan  info  t] supply [
    face/text: either e: pick x count: count + s [e/:index] [none]
  ] sl: slider 20x412 [s: value * length? x show l] return
  btn "Add"[y: maximum-of x repend/only x[1 + y/1/1 random now cp""]show l]
  btn "Delete" [r: do request-text/title"#" remove-each i x[i/1 = r]show l]
  btn "Find"[u: cp[] h: request-text foreach i x[if find v: form i h[append
    u v]] if not f: request-list""u[exit]s:(do first parse f"")- 1 show l]
  btn "Save" [attempt [save request-file/only/save/file %x.txt x]]
  btn "Load" [attempt [x: load request-file/only/file %x.txt show l]]
  key keycode [down] [sl/data: (s: s + 10)/ length? x show g]
]
```



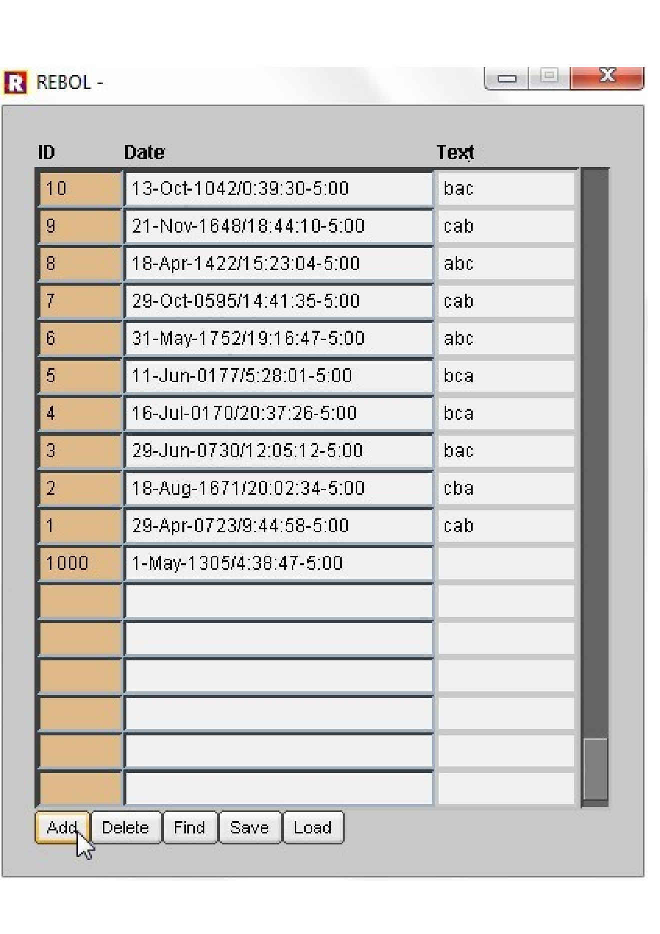
REBOL -
ID
Date
Text
1
29-Apr-0723/9:44:58-5:00
cab
2
18-Aug-1671/20:02:34-5:00
cba
3
29-Jun-0730/12:05:12-5:00
bac
4
16-Jul-0170/20:37:26-5:00
bca
5
11-Jun-0177/5:28:01-5:00
bca
6
31-May-1752/19:16:47-5:00
abc
7
29-Oct-0595/14:41:35-5:00
cab
8
18-Apr-1422/15:23:04-5:00
abc
9
21-Nov-1648/18:44:10-5:00
cab
10
13-Oct-1042/0:39:30-5:00
bac
11
18-Oct-0909/17:12:14-5:00
cab
12
5-Feb-1885/5:11:03-5:00
acb
13
8-Mar-0593/2:37:22-5:00
acb
14
10-Apr-0206/15:13:06-5:00
acb
15
3-May-0317/19:13:49-5:00
cba
16
5-Nov-1592/13:30:23-5:00
bca
17
7-Aug-1646/8:08:06-5:00
cba
Add
Delete
Find
Save
Load

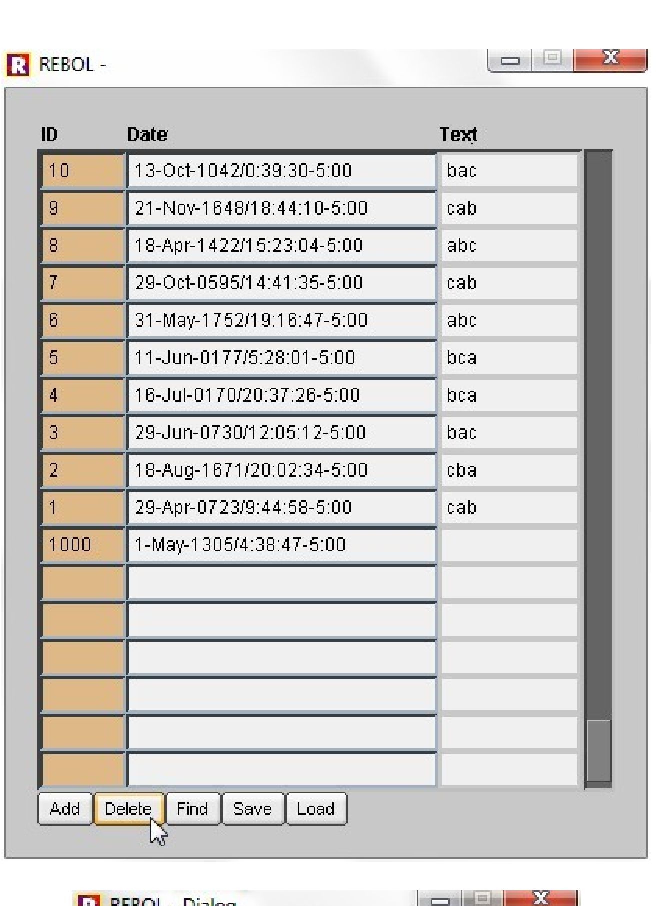


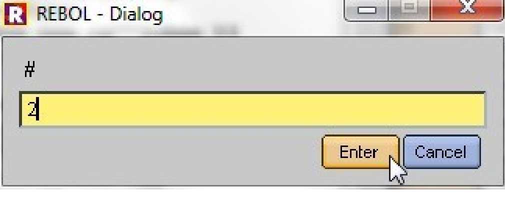
REBOL -
ID
Date
Text
999
8-Sep-0239/0:41:37-5:00
acb
998
7-Jul-0297/3:42:49-5:00
acb
997
13-Jan-0972/18:16:35-5:00
cab
996
18-Nov-0036/16:17:10-5:00
bac
995
27-Jun-0149/9:36:22-5:00
cab
994
21-May-1600/13:45:22-5:00
acb
993
14-Sep-0126/12:30:56-5:00
cba
992
12-Oct-0590/8:37:11-5:00
cba
991
10-May-1677/1:52:02-5:00
abc
990
29-Mar-0442/18:45:17-5:00
acb
989
16-Nov-1518/4:29:51-5:00
cba
988
11-Jun-1075/5:37:45-5:00
bca
987
12-Jun-0866/16:26:12-5:00
cba
986
27-May-1739/1:03:31-5:00
acb
985
1-Mar-1504/11:40:28-5:00
bca
984
12-Mar-1229/19:09:28-5:00
cab
983
28-Apr-0718/0:42:56-5:00
cab
Add
Delete
Find
Save
Load

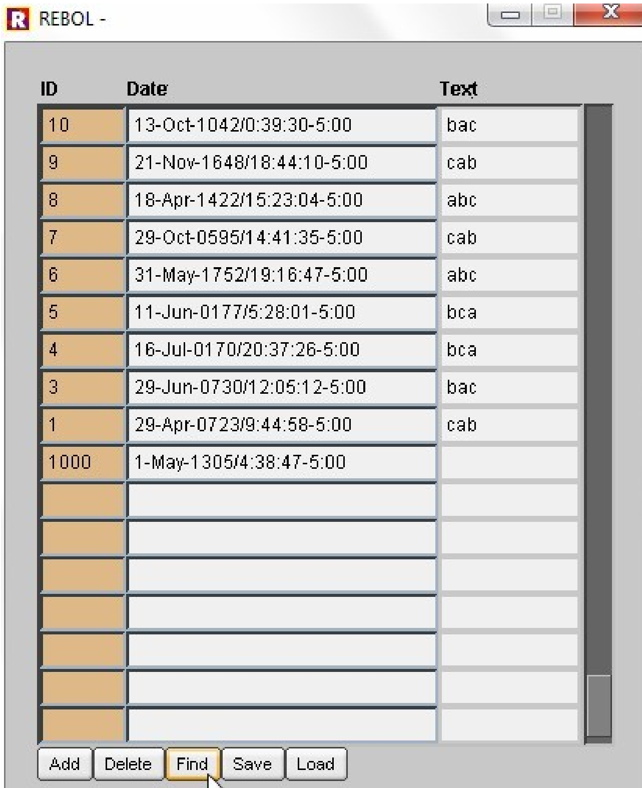

REBOL -
ID
Date
Text
999
8-Sep-0239/0:41:37-5:00
acb
998
7-Jul-0297/3:42:49-5:00
acb
997
13-Jan-0972/18:16:35-5:00
cab
996
18-Nov-0036/16:17:10-5:00
bac
995
27-Jun-0149/9:36:22-5:00
cab
994
21-May-1600/13:45:22-5:00
acb
993
14-Sep-0126/12:30:56-5:00
cba
992
12-Oct-0590/8:37:11-5:00
cba
991
10-May-1677/1:52:02-5:00
abc
990
29-Mar-0442/18:45:17-5:00
acb
989
16-Nov-1518/4:29:51-5:00
cba
988
11-Jun-1075/5:37:45-5:00
bca
987
12-Jun-0866/16:26:12-5:00
cba
986
27-May-1739/1:03:31-5:00
acb
985
1-Mar-1504/11:40:28-5:00
bca
984
12-Mar-1229/19:09:28-5:00
cab
983
28-Apr-0718/0:42:56-5:00
cab
Add
Delete
Find
Save
Load

REBOL - Dialog
Enter text below:
acbefg
Enter
Cancel

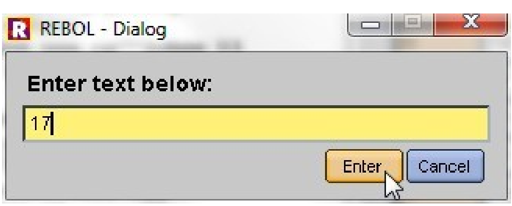

REBOL -
ID
Date
Text
10
13-Oct-1042/0:39:30-5:00
bac
9
21-Nov-1648/18:44:10-5:00
cab
8
18-Apr-1422/15:23:04-5:00
abc
7
29-Oct-0595/14:41:35-5:00
cab
6
31-May-1752/19:16:47-5:00
abc
5
11-Jun-0177/5:28:01-5:00
bca
4
16-Jul-0170/20:37:26-5:00
bca
3
29-Jun-0730/12:05:12-5:00
bac
2
18-Aug-1671/20:02:34-5:00
cba
1
29-Apr-0723/9:44:58-5:00
cab
1000
1-May-1305/4:38:47-5:00
Add
Delete
Find
Save
Load


REBOL -
ID
Date
Text
10
13-Oct-1042/0:39:30-5:00
bac
9
21-Nov-1648/18:44:10-5:00
cab
8
18-Apr-1422/15:23:04-5:00
abc
7
29-Oct-0595/14:41:35-5:00
cab
6
31-May-1752/19:16:47-5:00
abc
5
11-Jun-0177/5:28:01-5:00
bca
4
16-Jul-0170/20:37:26-5:00
bca
3
29-Jun-0730/12:05:12-5:00
bac
2
18-Aug-1671/20:02:34-5:00
cba
1
29-Apr-0723/9:44:58-5:00
cab
1000
1-May-1305/4:38:47-5:00
Add
Delete
Find
Save
Load

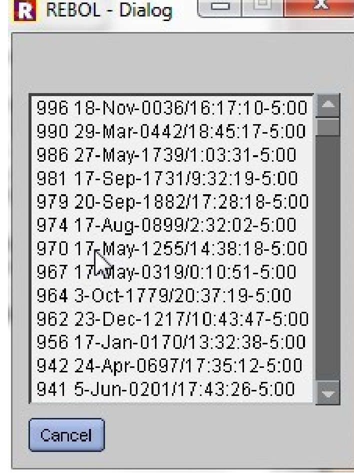

REBOL - Dialog
#
2
Enter
Cancel



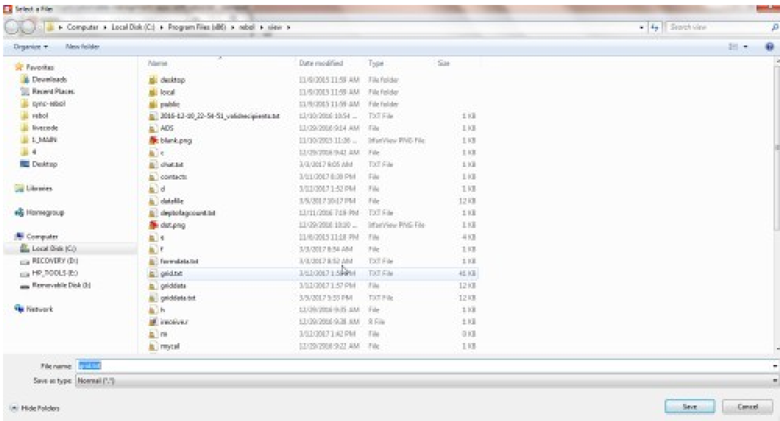
REBOL -
ID
Date
Text
10
13-Oct-1042/0:39:30-5:00
bac
9
21-Nov-1648/18:44:10-5:00
cab
8
18-Apr-1422/15:23:04-5:00
abc
7
29-Oct-0595/14:41:35-5:00
cab
6
31-May-1752/19:16:47-5:00
abc
5
11-Jun-0177/5:28:01-5:00
bca
4
16-Jul-0170/20:37:26-5:00
bca
3
29-Jun-0730/12:05:12-5:00
bac
1
29-Apr-0723/9:44:58-5:00
cab
1000
1-May-1305/4:38:47-5:00
Add
Delete
Find
Save
Load



REBOL - Dialog
Enter text below:
17
Enter
Cancel



REBOL - Dialog
996 18-Nov-0036/16:17:10-5:00
990 29-Mar-0442/18:45:17-5:00
986 27-May-1739/1:03:31-5:00
981 17-Sep-1731/9:32:19-5:00
979 20-Sep-1882/17:28:18-5:00
974 17-Aug-0899/2:32:02-5:00
970 17-May-1255/14:38:18-5:00
967 17-May-0319/0:10:51-5:00
964 3-Oct-1779/20:37:19-5:00
962 23-Dec-1217/10:43:47-5:00
956 17-Jan-0170/13:32:38-5:00
942 24-Apr-0697/17:35:12-5:00
941 5-Jun-0201/17:43:26-5:00
Cancel



Try using all the features in the app above:

1. Edit values in the 3rd column.
2. Click on each column header a few times to sort columns ascending and descending (notice that the key values are sorted numerically, the date values are sorted chronologically, and the text values are sorted alphabetically)
3. Add new rows to the data, and edit the added rows.
4. Delete rows.
5. Search for values in the data.
6. Save the edited data block to a file.
7. Close and restart the app and load the saved data file.

Also, try adjusting some of the default values in the app:

1. Create a larger or smaller number of random row values in the first line of the code.
2. Remove an existing column of data, and add some new columns of data.
3. Adjust the size, color, font and other properties of each row layout in the grid display.
4. Try adding new features such undo after a delete, or a search which can target individual columns, etc. (there features are actually available in the next grid example app).

The most important not-so-obvious thing to understand about this example, is that it's built from the same basic generic language components that all other Rebol apps are built from: word labels, block data structures and functions which manipulate those structures, loops, conditional evaluations, etc. The thought process, the language components, and the tooling needed to creating games, network apps, graphic and sound apps, web site apps, etc., with Rebol are all basically the same. The features of this grid display don't rely on some massive widget which comes as a monolithic 3rd party library, with functions and/or data structures unique to its use only. Instead, the grid components in this example are no more complicated than the previous 5 line example. The grid itself is a lightweight, fast, and versatile display tool which integrets perfectly with the basic language and data structure patterns used in every other sort of Rebol development. Everything else in this example is pure Rebol code, manipulating block data structures. The great thing about this is that, as you become more proficient at accomplishing any one sort of work with Rebol, you become also become proficient at performing many other types of work with Rebol. That's one of the greatest design points which makes Rebol so very different, from the ground up, than other development environments. The language and tooling design is incredibly light, versatile, integrated, and able to be applied universally to so many different problem domains, without the need for external libraries or more complex language structure.

If you want to add buttons, for example, to compute Sum, Average, and Maximum values of column 1 in the grid above, just use 'foreach' loops to collect and perform computations upon the first item in each sub-block in 'x':

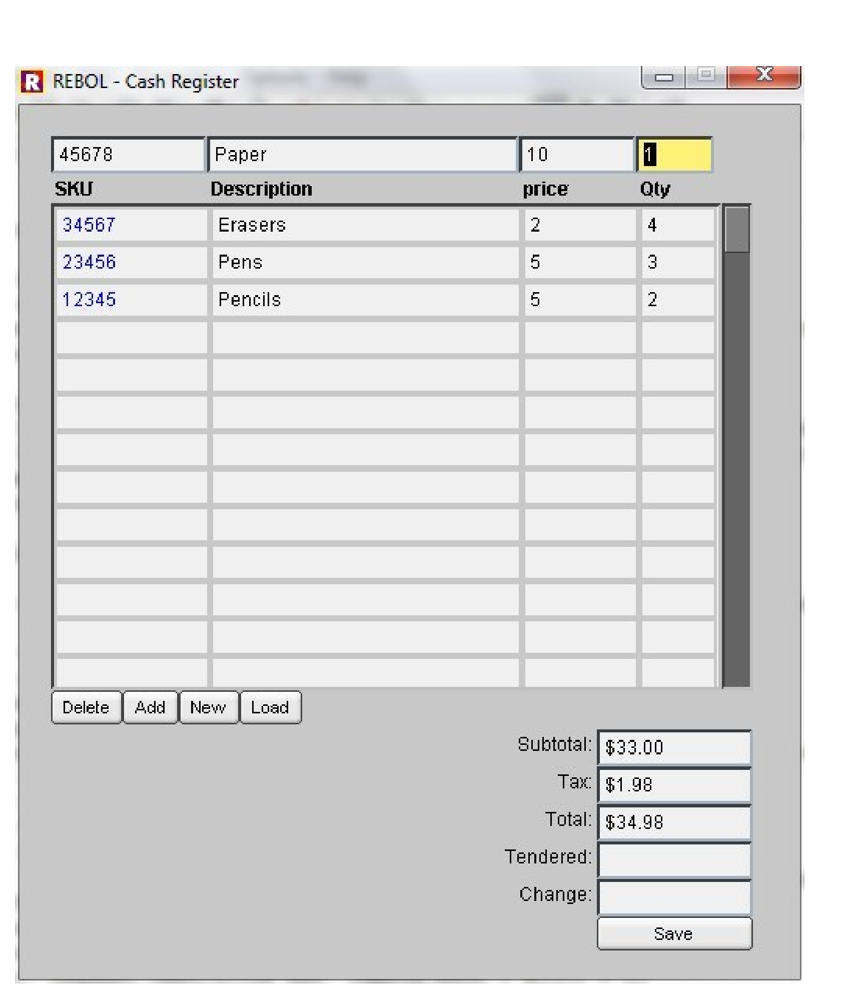
```
btn "Sum" [sm: 0 foreach i x [sm: sm + i/1] notify form sm]
btn "Avrg" [sm: 0 foreach i x [sm: sm + i/1] notify form sm / length? x]
btn "Max" [foreach i x [append y:[] i/1] notify form first maximum-of y]
```

8.3 Cash Register App Using the Basic Grid Features

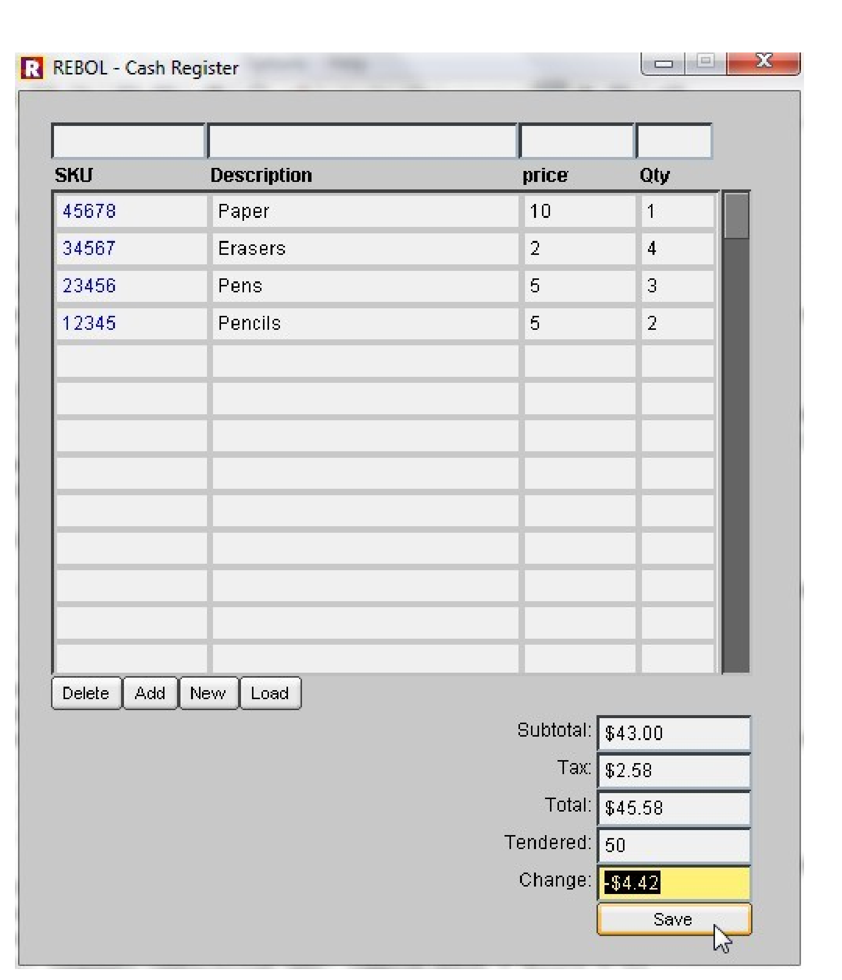
Here's a little cash register app built using many of the grid features above. It also demonstrates 1 way data in the grid can be formatted for printing (in this example, the formatted data is saved to an .html file and then opened with the browser, where it prints automatically):

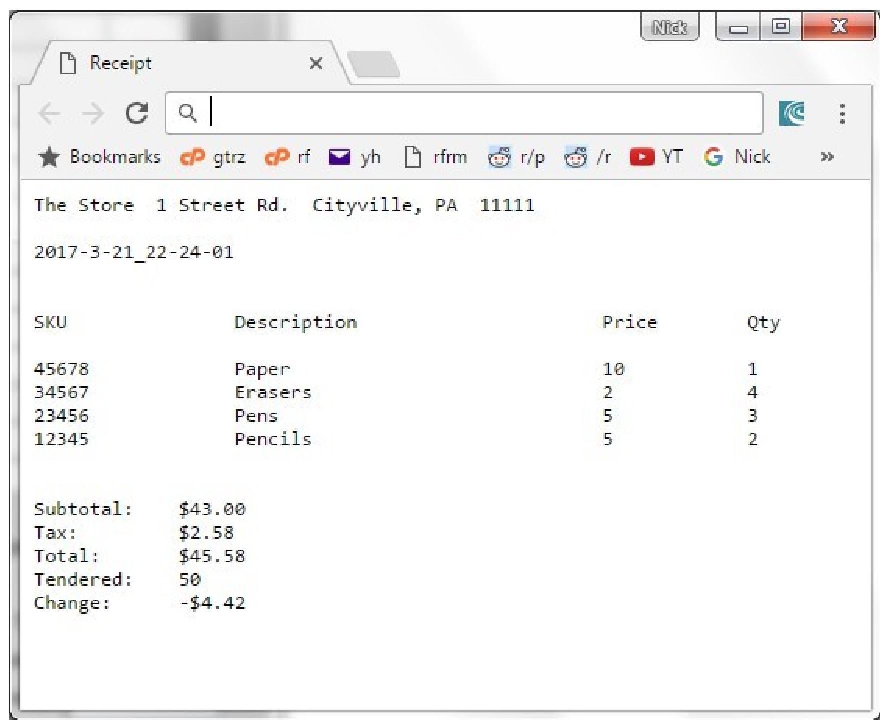
```
REBOL [title: "Cash Register"]
s: 0  asc: 1  x: copy []  tx: .06  call ""
pad: func [str len] [
  append copy/part head insert/dup tail cp str " " len len "    "
]
calculate: does [
  sum: $0 foreach i x [sum: sum + ((do i/3) * do i/4)]
  subtotal/text: sum  tax/text: (sum * tx)  total/text: sum * (1 + tx)
  show l clear-fields g show g focus sku
]
add-item: does  [
  if error? try [to-money copy price/text][alert "Price error" return]
  if error? try [to-decimal copy qty/text][alert "Qty error" return]
  insert/only head x reduce [
    cp sku/text cp desc/text cp price/text cp qty/text
  ]
  calculate
]
edit: func [f] [f/text: request-text/default f/text show l calculate]
srt: func [c] [sort/compare x func [a b] [(at a c)< at b c]
  if asc: not asc [reverse x] show l]
view g: layout [
  size 500x560  across  space 0x0
  style t text 200x24 edge [size: 2x2][edit face]
  sku: field 100  desc: field 200  price: field 75  qty: field 50  return
  h4 "SKU" 100 [srt 1]
  h4 "Description" 200 [srt 2]
  h4 75 "price" [srt 3]
  h4 50 "Qty" [sort 4]  return
  l: list 430x312 [across  space 0x0  t 100 blue t t 75 t 50] supply [
    face/text: either e: pick x count: count + s [e/:index] [none]
  ] sl: slider 20x312 [s: value * length? x show l] return
  btn "Delete" #"^~" [
    r: request-text/title"SKU" remove-each i x[i/1 = r]
    calculate
  ]
  btn "Add" #"^M" [add-item]
  btn "New" [if request "Delete Current Sale?" [x: copy [] calculate]]
  btn "Load" [if request "Delete Current Sale?" [
    attempt [x: load request-file/only  calculate]
  ]] return
  key keycode [down] [sl/data: (s: s + 10)/ length? x show g]
  key keycode [up] [sl/data: (s: s - 10)/ length? x show g]
  origin 300x400  style tot text 70 right style fld field 100
  tot "Subtotal:" subtotal: fld return
  tot "Tax:" tax: fld return
  tot "Total:" total: fld return
  tot "Tendered:" tendered: fld 100 [
    if error? try [chng/text: do total/text - do face/text][
       alert "Tendered Error"
    ]
  ] return
  tot "Change:" chng: fld 100 return
  tot btn 100 "Save" [
   if error? try [
      save fl: to-file rejoin [
        now/year "-" now/month "-" now/day "_"
        replace/all form now/time ":" "-"
      ] x
      rcpt: copy {<html><head><title>Receipt</title>
<script type="text/javascript">
<!--
  function printthispage() {
  window.print();
```

```
  }
  //-->
</script></head><body onload="printthispage()"><pre>}
      col1: 15  col2: 30  col3: 10  col4: 6
      append rcpt rejoin [
        "The Store  1 Street Rd.  Cityville, PA  11111" "^/^/"
        fl "^/^/^/"
        pad "SKU" col1 pad "Description" col2
        pad "Price" col3 pad "Qty" col4  "^/^/"
      ]
      foreach i x [
        append rcpt rejoin [
          pad i/1 col1 pad i/2 col2 pad i/3 col3 pad i/4 col4 newline
        ]
      ]
      append rcpt rejoin [
        newline newline pad "Subtotal:" 10 subtotal/text
        newline pad "Tax:" 10 tax/text
        newline pad "Total:" 10 total/text
        newline pad "Tendered:" 10 tendered/text
        newline pad "Change:" 10 chng/text
        </pre>
      ]
      write prn: join fl ".html" rcpt
      ; call/show join "notepad " to-local-file prn
      browse prn
      x: copy [] focus sku clear-fields g show g show l calculate
    ][
      alert "Error Saving"
    ]
  ]
  do [focus sku]
]
```



REBOL - Cash Register
45678
Paper
10
1
SKU
Description
price
Qty
34567
Erasers
2
4
23456
Pens
5
3
12345
Pencils
5
2
Delete
Add
New
Load
Subtotal:
$33.00
Tax:
$1.98
Total:
$34.98
Tendered:
Change:
Save


REBOL - Cash Register
SKU
Description
price
Qty
45678
Paper
10
1
34567
Erasers
2
4
23456
Pens
5
3
12345
Pencils
5
2
Delete
Add
New
Load
Subtotal:
$43.00
Tax:
$2.58
Total:
$45.58
Tendered:
50
Change:
-$4.42
Save

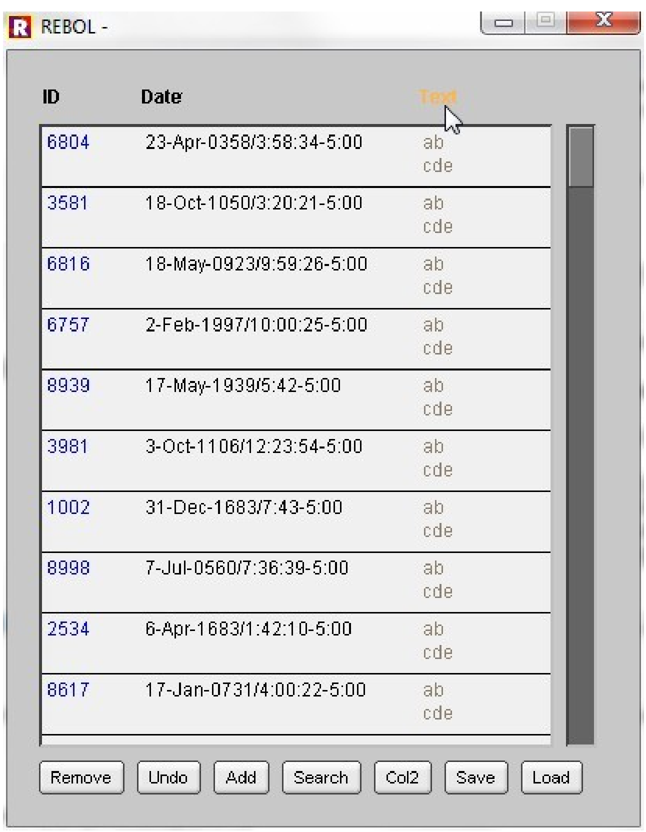


8.4 Even More Features

Here's a more complete 'list' widget example which demonstrates how to implement most of the commonly needed features in grid displays. Notice:

1. the black line in the otherwise non-grid look of each row.
2. the delete key keyboard shortcut for the remove operation, as well as both the up and down arrow key scroll features.
3. the column specific search function (it only searches within the column specified by the user, for the search term entered by the user).
4. the larger text-requestor function (request-big) (which allows you to edit columns which contain multiline data).
5. the 'undo' feature, which can be used to reverse the most recent row delete operation.
6. the improved 'add' feature, which allow the user add a new row of values at the currently selected position.
7. the 'col2' feature, which demonstrates how to collect values from a single column in the grid.

You can copy/paste and experiment with the given code to understand how each feature is created:




```
REBOL [title: "GRID: SAVE/LOAD, ADD/REMOVE, UNDO, EDIT, SORT, FIND, KEYS"]
sp: 0   asc: false   mx: cnt: 9999
repeat i cnt [repend/only x:[] [i random now join random"ab"random"cde^/"]]
scrl: func [arg] [sp: sp + arg  sl/data: sp / length? x  show g]
srt: func [f] [
  sort/compare x func[a b][(at a f) < (at b f)]
  if asc: not asc [reverse x] show li
]
do [request-big: do replace replace replace mold :request-text
  {field 300} "area 400x300" {with [flags: [return]]} "" {194} "294"]
updt: func [f] [f/text: request-big/default f/text show li]
view center-face g: layout [
  across h4 "ID" 55 [srt 1] h4 "Date" 170 [srt 2] h4 "Text" [srt 3] return
  li: list 330x400 [
    across
    text 55 blue [cnt: face/data]
    text 170
    text 80x30 rebolor [updt face]
    return box black 400x1
  ] supply [
    if none? e: pick x count: count + sp [face/text: none exit]
    face/text: pick e index   face/data: count
  ]
  sl: slider 20x400 [sp: value * length? x  show li]  return
  btn #"^~" "Remove" [ud: first at x cnt  remove at x cnt  show li]
  btn "Undo" [if value? 'ud [insert/only at x cnt ud unset 'ud show g]]
  btn "Add" [
    mx: mx + 1
    insert/only at x ++ cnt + 1 reduce[mx random now random "abcd"] show li
  ]
  btn "Search" [
    if find reduce[p: request-list"Field:"[1 2 3]h: request-text]none[exit]
```

```
    u: copy[] foreach i x [foreach [j k l] i [
      if find pick reduce [form j form k form l] p h [append u form i]
    ]] if empty? u [notify "Not found" exit] fnd: request-list "" u
    repeat i length? x [if (fnd = form x/:i)[sp: i - 1 break]] show g
  ]
  btn "Col2" [foreach [i j] x [append y:[] i/2] editor y]
  btn "Save" [attempt [save to-file request-file/save/file %grid.txt x]]
  btn "Load" [attempt [x: load request-file/only/file %grid.txt show li]]
  key keycode [down][scrl 10]  key keycode [up][scrl -10]
]
```

Here's an example of the CRUD app demonstrated earlier, with the generated grid display replaced by a list widget:

```
REBOL [title: "Generic CRUD & List Grid Combined + Search/Add/Delete/IEF"]
n: [foreach i[a b c][set-face get i""]focus a]  s: [save %d d  notify"ok"]
x: [do z repend d[a/text b/text c/text]do n]  ief: :insert-event-func
y: [attempt[r: copy/part(find/skip d (request-list"" extract d 3)3)3
  repeat i length? j:[a b c][set-face get j/:i r/:i]]]  ; change-dir %/c/
z: [remove-each[i j k]d[i = a/text]]  write/append %d ""  d: load %d
view center-face layout [
  a: field  b: field  c: area  across
  btn "New"  [do n]
  btn "Save (delete)" [do x do s][do z do n]
  btn "Edit" [do y]
]
sp: 0  cnt: 1  x: copy [] foreach [j k l]d [append/only x reduce [j k l]]
rq: do replace replace replace mold :request-text
  {field 300} "area 400x300" {with [flags: [return]]} "" {194} "294"
updt: func [f][f/text: rq/default f/text show li]
sv: does [foreach i x [append dd:[] i] save %d dd]
ief [e: :event either e/type = 'close [if request "Save?" [sv] e][e]]
view center-face g: layout [
  across
  li: list 620x400 [
    across text 100 [cnt: face/data updt face]
    text 80  [updt face]
    text 400x40 [updt face]
    return box black 620x1
  ] supply [
    if none? e: pick x count: count + sp [face/text: none exit]
    face/text: pick e index  face/data: count
```

```
  ]
  sl: slider 20x400 [sp: value * length? x  show li]  return
  btn #"^~" "Remove" [remove at x cnt  show li]
  btn "Add" [insert/only at x ++ cnt + 1 reduce[copy""copy""copy""]show li]
  btn "Search" [
    if find reduce[p: request-list"Field:"[1 2 3]h: request-text]none[exit]
    u: copy[] foreach i x [foreach [j k l] i [
      if find pick reduce [form j form k form l] p h [append u form i]
    ]] if empty? u [notify "Not found" exit] fnd: request-list "" u
    repeat i length? x [if (fnd = form x/:i)[sp: i - 1 break]] show g
  ]
  btn "Data" [editor x]
]
```

9. Parse

The 'parse' function is one of Rebol's greatest strengths. It can be used to split strings of text, perform complicated text search and match operations, validate text input, and it's even the basis of Rebol's amazing dialecting capabilities (the ability to create powerful little domain-specific languages...).

This example simply splits the text in a given string:

```
parse "apple orange pear" none      ; simple splitting, together with
parse "apple, orange, pear" none    ; read/lines, is useful with CSV files
```

This example prints the second comma-separated item in each line of a download Paypal account file:

```
foreach line at read/lines http://re-bol.com/Download.csv 2 [
  print second parse/all line ","
]
```

This example expands on the code above to perform time evaluations on the downloaded data. It prints item #8 of the comma separated values, only if the time value (item #2) on the line is between midnight and noon. This is a very powerful and useful template for the kind of code which is used often in CRUD apps:

```
foreach line at read/lines http://re-bol.com/Download.csv 2 [
    time: to-time second row: parse/all line ","
    if (time >= 0:00am) and (time <= 12:00pm) [print row/8]
]
ask "Press [ENTER] to Continue..."
```

This example separates the letter of each word in a list, at the vowel characters:

```
foreach i ["apple" "orange" "pear"] [probe parse i "aeiou"]
```

This example uses the parse technique above to generate Pig Latin from a sentence input by the user. Don't worry about understanding every bit of code in this example at this point, just watch it run:

```
do [foreach t parse ask "Type a sentence:^/^/" "" [
    x: parse t "aeiou" prin rejoin [
        either p: find/match t x/1[p][t]x/1 either p["ay"]["hay"]" "
    ]
]]
```

The 'parse' pattern in the following example is common. It searches for every occurrence of text between 'src="' and '"', and adds those found bits of text to a block. This is another commonly useful code template to memorize:

```
html: read http://musiclessonz.com/ScreenShots.html
parse html [any [thru {src="} copy l to {"} (append pics: [] l)] to end]
editor pics    ; this pattern is used to search text, code, etc.
```

This example demonstrates the 'parse' pattern above in an app that creates mad libs from text input by the user. Again, it's ok if you don't completely understand this example, just watch the parse example in action:

```
t: {<name> found a <thing>. <name> ate the <adjective> <thing>.}
view layout [a: area wrap t  btn "Make Madlib" [x: a/text unview]]
parse x [any [to "<" copy b thru ">" (append w: [] b)] to end]
foreach i unique w [replace/all x i ask join i ": "]  notify x
```

This example uses 2 'parse' patterns to extract the IP address information from a web page (interestingly, that web page is a Rebol script running on a web server).

```
parse read http://guitarz.org/ip.cgi [thru <title> copy w to </title>]
print wan: last parse w none
```

This example demonstrates 'parse' doing some more complex conditional and block operations, to remove comments from a chosen code file:

```
code: read to-file request-file
parse/all code [any [
    to #";" begin: thru newline ending: (
        remove/part begin ((index? ending) - (index? begin))) :begin
    ]
] editor code  ; all comments removed
```

This example demonstrates how 'parse' can be used to validate user input in GUIs:

```
nums: charset "0123456789" alfs: charset [#"a" - #"z" #"A" - #"Z"] call ""
validate: func [f rule] [if not parse f/text rule [attempt [
  insert s: open sound:// load %/c/windows/media/chord.wav wait s close s
]]]
view layout [
  f1: field "12345678" [validate face [8 nums]]
  field "1234" [validate face [4 nums]]
  field "(555)123-2345" [validate face ["(" 3 nums ")" 3 nums "-" 4 nums]]
  field "me@url.com" [validate face [some alfs "@" some alfs "." 3 alfs]]
  do [focus f1]
]
```

Parse techniques can be extremely powerful, and they're one of the main reasons that developers choose to use Rebol. Parse replaces the need for 'Regular Expressions' which are used in almost all other programming languages. Parse code is much easier to read, and it generally performs faster than regular expression implementations.

10. Some Additional App Examples to Study

All the following example apps are presented to help demonstrate additional common techniques used in CRUD apps. Look for word labels, blocks, functions, loops and conditional evaluations, GUI layouts, etc. If

you can't understand how some part of any of these apps work, you can read line-by-explanations in the tutorials at http://business-programming.com and http://re-bol.com/rebol_quick_start.html.

10.1 Cash Register

Here's a general quick description of how the app works. Edit the code to add your own features!

1. A GUI layout is created with fields for cashier, item, price subtotal, tax, and total values, as well as an area widget to display the list of added items, and a button to save the order.
2. Most of the operational code is run when a value is entered into the price field. An error check assures that a proper money value has been entered. The item and price values are then added to the area widget display, the item fields are cleared, then the computations for subtotal, tax, and total money values are performed and the results displayed in the appropriate field widgets.
3. When the user clicks the save button, the data in the area widget is formatted in a way which enables it to be saved (appended) to a file, and then the display is cleared.

A great exercise would be to rewrite this app using the list widget techniques you learned in the previous section.

```
REBOL [title: "Minimal Cash Register"]
view gui: layout [
    style fld field 80  across
    text "Cashier:"   cashier: fld
    text "Item:"      item: fld
    text "Price:"     price: fld [
        if error? try [to-money price/text] [notify "Price error" return]
        append a/text reduce [mold item/text tab price/text newline]
        item/text: copy "" price/text: copy ""
        sum: 0 foreach [item price] load a/text[sum: sum + to-money price]
        subtotal/text: form sum
        tax/text: form sum * .06
        total/text: form sum * 1.06
        focus item show gui
    ]
    return  a: area 600x300  return
    text "Subtotal:"   subtotal: fld
    text "Tax:"        tax: fld
    text "Total:"      total: fld
    btn "Save" [
        items: replace/all (mold load a/text) newline " "
        write/append %sales.txt rejoin [
            items newline cashier/text newline now/date newline
        ]
        clear-fields gui  a/text: copy ""  show gui
    ]
]
```

10.2 Cash Register Report

This example reads the data from the file saved in the previous example app, and uses several 'foreach' loops and some conditional evaluations and math to produce a daily sales report:

```
REBOL [title: "Daily Cash Register Report"]
sales: read/lines %sales.txt  sum: $0
foreach [items cashier date] sales [
    if now/date = to-date date [
        foreach [item price] load items [
            sum: sum + to-money price
        ]
    ]
] notify rejoin ["Total sales today: " sum]
```

10.3 Web Chat

This example allows an unlimited number of users to participate in a text chat, using a simple text file stored on a web server:

```
REBOL [title: "Group Text Chat, Desktop App"] ; connects w/ web chat above
url: ftp://user:pass@site.com/public_html/chat.txt   ; FTP server required
write/append url ""
name: copy ask "^LName:  "
forever [
    notes: copy read url
    message: ask rejoin [newpage notes "Message:  "]
    if message = "erase" [write url ""]
    if message <> "" [
        write/append url rejoin [
            now " (" name "):  " message "^/^/"
        ]
    ]
]
```

10.4 Simple File Sharer

```
REBOL [title: "Simple File Sharer"] ; instantly create pages for new users
login: request-text/default "ftp://user:pass@site.com/public_html/files/"
filename: join request-text/default "links" ".html"
html: copy {}
foreach f request-file [
    print file: last split-path f
    write/binary to-url join login file read/binary f
    append html rejoin [
        {<a href="} file {">} file {</a><br>} newline
    ]
]
write/append html-file: to-url join login filename html
editor html-file     ; edit and re-save page, if desired
browse join "http://" replace next find html-file "@" "public_html/" ""
```

10.5 Simple Full Screen Slide Presentation

```
REBOL [title: "Simple Full Screen Slide Presentation"]
slides: [
    [
        at 0x0 box system/view/screen-face/size white [unview]
        at 20x20 h1 blue "Slide 1"
        box black 2000x2
        text "This slide takes up the full screen."
        text "Adding images is easy:"
        image logo.gif
        image stop.gif
        image info.gif
        image exclamation.gif
        text "Click anywhere on the screen for next slide..."
        box black 2000x2
    ]
    [
        at 0x0 box system/view/screen-face/size effect [
            gradient 1x1 tan brown
        ] [unview]
        at 20x20 h1 blue "Slide 2"
        box black 2000x2
        text "Gradients and color effects are easy in REBOL:"
        box effect [gradient 123.23.56 254.0.12]
```

```
        box effect [gradient blue gold/2]
        text "Click anywhere on the screen to close..."
        box black 2000x2
    ]
    [
        at 0x0 box 600x400 [unview]
        at 20x20
        text "This screen is smaller, and as simple as can be"
        text "Click anywhere on the screen to close..."
    ]
]
foreach slide slides [
    view/options center-face layout slide 'no-title
]
```

10.6 Simple Encrypt/Decrypt Utility

```
REBOL [title: "Simple Encrypt/Decrypt Utility"]
view layout [
    across  f: field  btn "Select File" [f/text: request-file/only show f]
    return  f2: field "(enter password)"  btn "Encrypt" [
        save request-file/only/save/title/file
            "Enter encrypted file save name:" "" to-file f/text
            to-binary encloak read to-file f/text f2/text  alert "Saved"
    ]
    btn "Decrypt" [editor to-string decloak load to-file f/text f2/text]
]
```

10.7 More Advanced To-Do List

```
REBOL [title: "To Do List"]   ; with adjustable priorities and screen shot
view/new gui: layout [
    t: text-list 550x400
    across
    f: field [if not find t/data v: value [append t/data copy v] focus f]
    btn "Del"   [remove find t/data t/picked]
    btn "Up"    [attempt [move/to z: find t/data t/picked (index? z) - 1]]
    btn "Down"  [attempt [move/to z: find t/data t/picked (index? z) + 1]]
    btn "Sort"  [sort t/data  show t]
    btn "Load"  [attempt [t/data: load request-file/file/only %todo.txt]]
    btn "Save"  [attempt [
        save request-file/save/file/only %todo.txt t/data
        notify "Saved"
    ]]
    btn "Print" [save/png %todo.png to-image t  browse %todo.png]
]
forever [wait .1  show gui  if not viewed? gui [quit]]
```

10.8 Recursive Text in Files Search

```
REBOL [title: "Recursive Text in Files Search"]   ; search all sub-folders
phrase: request-text/title/default "Text to Find:" "the"
change-dir start: request-dir/title "Folder to Start In:"
found-list: ""
recurse: func [folder] [
    foreach item (read folder) [
        either dir? item [
            change-dir item   recurse %.\   change-dir %..\
        ][
            if find (read to-file item) phrase [
                print rejoin [{"} phrase {" found in:  } what-dir item]
```

```
                  found-list: rejoin [found-list newline what-dir item]
              ]
          ]
      ]
]
print rejoin [{SEARCHING for "} phrase {" in } start "...^/"]
recurse %.\   print "^/DONE^/"   editor found-list
```

10.9 Little Blogger

```
REBOL [title: "Little Blogger"]
url: ftp://user:pass@url.com/public_html/myblog.html
view layout[
    a: area 400x300 "Today I ..." across
    btn 100 "PUBLISH" [
        write/append url ""
        page: rejoin ["<pre>" now "^/^/" get-face a "</pre><hr>" read url]
        either er: error? try [write url page] [notify er] [browse url]
    ]
    btn "Browse" [browse replace replace url {ftp}{http}{public_html/}{}]
    btn "Edit" [editor url]
]
```

10.10 Console Calendar Printer

```
REBOL [title: "Print Console Calendar for Chosen Year"]
if "" = y: ask "Year (ENTER for current):^/^/" [prin y: now/year]
foreach m system/locale/months [
  prin rejoin ["^/^/      " m "^/^/ "]
  foreach day system/locale/days [prin join copy/part day 2 " "]
  print "" f: to-date rejoin ["1-"m"-"y]  loop f/weekday - 1 [prin "   "]
  repeat i 31 [
    if attempt [c: to-date rejoin [i"-"m"-"y]][
      prin join either 1 = length? form i ["  "][" "] i
      if c/weekday = 7 [print ""]
    ]
  ]
] print "^/"  halt
```

10.11 Continuously Echo Clipboard to File

```
REBOL [title: "Continuously Echo Clipboard Contents to File"]
x: copy {}   print "started..."
forever [if x <> y: read clipboard:// [
    write/append %clipboard.txt y  probe y  x: copy y
]]
```

10.12 FTP Tool

```
REBOL [title: "FTP Tool"] ; visually list and edit any file on FTP servers
view  layout [
    f: field 600 "ftp://user:pass@site.com/public_html/" [
        either dir? to-url value [
            t/data: sort read to-url value  show t
        ][
            editor to-url value
        ]
    ]
```

```
    t: text-list 600x400 [editor to-url join f/text value]
]
```

10.13 More GUI Examples

```
REBOL [title: "Changing Panes (panels in a GUI layout)"]
gui: layout [
    b: box 400x200
    btn "Switch to Text List"    [b/pane: pane2 show b]
    btn "Switch to Fields"       [b/pane: pane1 show b]
]
pane1: layout/tight [field 400 field 400 area 400]
pane2: layout/tight [text-list 400x200 data system/locale/days]
b/pane: pane1
view gui


REBOL [title: "Choice Button Menu Example"]
menu1: [
    "Options" []
    "_______________^/" []
    "    Load"  [attempt [a/text: read request-file/only show a]]
    "    Save"  [attempt [write request-file/only a/text]]
    "_______________^/" []
    "    Quit"  [quit] ""[]
]
view layout [
    choice 120x20 aqua black left data (extract menu1 2) [
        do select menu1 value   face/text: face/texts/1   show face
    ]
    a: area 500x400
]


REBOL [title: "Generic GUI Calculator"]
view layout [
     f: field 225x50 font-size 25  across
     style b btn 50x50 [append f/text face/text show f]
     b "1"  b "2"  b "3"  b " + "  return
     b "4"  b "5"  b "6"  b " - "  return
     b "7"  b "8"  b "9"  b " * "  return
     b "0"  b "."  b " / "  b "="  [
         attempt [f/text: form do f/text show f]
     ]
]
```

10.14 Network Apps (HTML Server and Handler, Text Messagers, Email, VOIP, Etc.)

See http://business-programming.com for more information about how to send data across network and other ports.

```
REBOL [title: "TCP Text Messager"]  ; (run on any 2 network-connected PCs)
view layout [ across
    q: btn "Serve"[focus g p: first wait open/lines tcp://:8 z: 1]text"OR"
    k: btn "Connect"[focus g p: open/lines rejoin[tcp:// i/text ":8"]z: 1]
    i: field form read join dns:// read dns://  return
    r: area rate 4 feel [engage: func [f a e][if a = 'time and value? 'z [
        if error? try [x: first wait p] [quit]
        r/text: rejoin [x "^/" r/text] show r
    ]]] return
    g: field "Type message here [ENTER]" [insert p value  focus face]
]
```

```
REBOL [title: "HTML Form Server & HANDLER"] ; STANDALONE, no Apache needed
port: open/lines tcp://:80   browse http://localhost?   print ""
forever [
    p: port/1 attempt [    ; SERVES form, PRINTS & SAVES responses to file
        probe x: decode-cgi replace next find p/1 "?" " HTTP/1.1" ""
        write-io p h:  {HTTP/1.0 200 OK^/Content-type: text/html^/^/
        <HTML><BODY><FORM action="localhost">
            Name:<input type="text" name="name"><input type="submit">
        </FORM></BODY></HTML>} length? h  write/append %f join x/2 newline
    ] close p
]


REBOL [Title: "UDP Group Chat"]    ; NO IP ADDRESSES required for any user
net-in: open udp://:9905   net-out: open/lines udp://255.255.255.255:9905
set-modes net-out [broadcast: on]
name: request-text/title "Your name:"
gui: view/new layout [
    a1: area wrap rejoin [name ", you are logged in."]
    f1: field
    k1: at 0x0 key #"^M" [
        if f1/text = "" [return]
        insert net-out rejoin [name ":  " f1/text]
    ]
] forever [
    focus f1
    received: wait [net-in]
    if not viewed? gui [quit]
    insert (at a1/text 1) copy received  show a1
]


REBOL [title: "Email"] ; account configuration, GUI send, import/read mail
system/user/email:  you@url.com     ss:  system/schemes
ss/default/host:  "smtp.url.com"    ss/default/user:   "username"
p: ss/pop/host:  pop://url.com      ss/default/pass:   "password"
view layout [
    text "Address:"   f: field "joe@url.com"
    text "Message:"   a: area
    btn "Send" [
        alert either attempt[send to-email f/text a/text]["Sent"]["Error"]
    ]
    btn "Read" [                 ; this displays 5000 characters by default
        foreach m read p [y: import-email m  ask copy/part y/content 5000]
    ]
]
```

See http://re-bol.com/rebol-multi-client-databases.html to learn about how to create network enabled multi-user data management apps which entirely eliminate the need for third party database systems, all using pure Rebol code:

```
REBOL [title: "** MULTI-USER ** Generic Network App Data SERVER"]
print "waiting..."
p: open/lines tcp://:55555
forever [
    if error? er: try  [
        probe cmmds: load data: first wait con: first wait p
        probe rslt: mold do cmmds insert con rslt  close con  true
        ; save %d d   ; commit data every s requests, or on timer, etc.
    ] [
        er: disarm :er
        print ne: rejoin [form now "^/" mold cmmds "^/" er "^/" "^/"]
        write %server-err.txt ne  close con
    ]
]
```

```
REBOL [title: "** MULTI-USER ** CRUD Database CLIENT App"]
ip: "localhost"  ; set this to the IP address of server machine
se: func [str /local er p st d n][      ; paste this generic function
  st: trim/lines str
  if error? er: try [
    p: open/lines rejoin[tcp:// ip ":55555"]
    insert p mold st d: first n: wait p close n close p return d
  ][
    er: disarm :er
    write/append %err.txt rejoin[now"^/"st"^/"er"^/^/"]
    alert"*Network Error* Try Again" none
  ] ; send a "to-block" in the se command when returning a block
]   ; send a 1 at the end of any se command that only affects server data
update: does [d: load se{to-block d}]  ; run after updating server data
n: [foreach i[a b c][set-face get i""]focus a]
s: [se rejoin["save %d " mold d " 1"] alert"ok"]
x: [do z  se rejoin[{repend d [} mold a/text mold
  b/text mold c/text {] 1}]  update  do n]
y: [attempt[r: copy/part(find d: load se{to-block d}request-list""
  extract d 3)3  repeat i length? j:[a b c][set-face get j/:i r/:i]]]
z: [se rejoin[{remove-each[i j k]d[i = }mold a/text{] 1}] update]
se{write/append %d "" 1} se{d: load %d 1} update
view g: layout[a: field b: field c: area across btn"New"[do n][do z do n]
  btn"Save"[do x] btn"Edit"[do y] btn"COMMIT"[do s] btn"Raw"[editor %d]
]
```

10.15 Web Site (CGI) App Examples

The ability to use web servers to interact with users is very powerful, especially when combined with the ability to shared data between native Rebol apps. See http://re-bol.com/rebol_quick_start.html and http://business-programming.com for more information about how to create Rebol apps that run on a web server.

```
#!./rebol -cs   ; location of Rebol interpreter
REBOL [title: "CGI Photo Album"] ; Web App - install on your Apache Server
print {content-type: text/html^/}
print {<HTML><HEAD><TITLE>Photos</TITLE></HEAD><BODY>}
folder: read %./
count: 0
foreach file folder [
    foreach ext [".jpg" ".gif" ".png" ".bmp"] [
        if find file ext [
            print rejoin [{<BR><CENTER><img src="} file {"></CENTER>}]
            count: count + 1
        ]
    ]
]
print rejoin [{<BR>Total Images: } count {</BODY></HTML>}]


#!./rebol -cs   ; location of Rebol interpreter
REBOL [title: "CGI Group Chat"]  ; Web App - install on your Apache Server
print {content-type: text/html^/}
url: %./chat.txt
write/append url ""
submitted: decode-cgi read-cgi
if submitted/2 = "erase" [write url ""]
if submitted/2 <> none [
    write/append url rejoin [
        now " (" submitted/2 "):  " submitted/4 "^/^/"
    ]
]
notes: copy read url
print rejoin [
    "<pre>" notes "</pre>"
```

```
    {<FORM METHOD="POST">
        Name:<br>
        <input type=text size="65" name="username"><br>
        Message:<br>
        <textarea name=message rows=5 cols=50></textarea><br>
        <input type="submit" name="submit" value="Submit">
    </FORM>}
]
```

10.16 DLL/Shared Lib Examples

The ability to use DLLs and libraries created by other developers, as well as the native API of the operating system, enables you to accomplish goals which aren't possible with the built-in capabilities of any language/toolset. See http://business-programming.com for more information about how to access DLL and shared libraries in Rebol apps.

```
REBOL [title: "USPS Intelligent Mail Encoder"]
unless exists? %usps4cb.dll [
    write/binary %usps4cb.dll read/binary http://re-bol.com/usps4cb.dll
]
GEN-CODESTRING: make routine! [
    t [string!]  r[string!]  c [string!]  return: [integer!]
]  load/library request-file/only/file %usps4cb.dll "USPS4CB"
t: request-text/title/default "Tracking #:" "00700901032403000000"
r: request-text/title/default "Routing #:" "55431308099"
GEN-CODESTRING t r (make string! 65)
alert first second first :GEN-CODESTRING


REBOL [title: "VOIP Intercom, for MS PCs"] do [write %ireceive.r {REBOL []
if error? try [port: first wait open/binary/no-wait tcp://:8] [quit]
wait 0  speakers: open sound://
forever [
    if error? try [mark: find wav: copy wait port #""] [quit]
    i: to-integer to-string copy/part wav mark
    while [i > length? remove/part wav next mark] [append wav port]
    insert speakers load to-binary decompress wav
]} launch %ireceive.r
lib: load/library %winmm.dll
mci: make routine! [c [string!] return: [logic!]] lib "mciExecute"
if (ip: ask "Connect to IP (none = localhost):  ") = "" [ip: "localhost"]
if error? try [port: open/binary/no-wait rejoin [tcp:// ip ":8"]] [quit]
mci "open new type waveaudio alias wav"
forever [
    mci "record wav"  wait 2  mci "save wav r"  mci "delete wav from 0"
    insert wav: compress to-string read/binary %r join l: length? wav #""
    if l > 4000 [insert port wav]  ; squelch (don't send) if too quiet
]]
```

Using libraries created by other Rebol developers can also be a big time saver:

```
REBOL [title: "Data Grid Library Example"]
headers: ["Column1" "Column2"]
x: data: [
     ["Item 01-01" "Item 02-01"]
     ["Item 01-02" "Item 02-02"]
     ["Item 01-03" "Item 02-03"]
     ["Item 01-04" "Item 02-04"]
     ["Item 01-05" "Item 02-05"]
]
do http://re-bol.com/gridlib.r


REBOL [title: "Q-Plot Lib Exploded Pie Chart Example"]
```

```
if not exists? %q-plot.r [write %q-plot.r read http://re-bol.com/q-plot.r]
do %q-plot.r
view center-face quick-plot [
    400x400
    pie [10 3 6 1 8] labels [A B C D E] explode [3 5]
    title "Exploded Sections C and E" style vh2
]


REBOL [title: "SQLITE Example"]
unless exists? %sqlite3.dll [
    write/binary %sqlite3.dll read/binary http://re-bol.com/sqlite3.dll
]
unless exists? %sqlite.r [
    write %sqlite.r read http://re-bol.com/sqlite.r
]
do %sqlite.r
db: connect/create %contacts.db
SQL "create table contacts (name, address, phone, birthday)"
SQL {insert into contacts values
    ('"John Doe"', '"1 Street Lane"', '"555-9876"', '"1967-10-10"')
}
data: [
    "John Smith" "123 Toleen Lane" "555-1234" "1972-02-01"
    "Paul Thompson" "234 Georgetown Pl." "555-2345" "1972-02-01"
    "Jim Persee" "345 Portman Pike" "555-3456" "1929-07-02"
    "George Jones" "456 Topforge Court" "" "1989-12-23"
    "Tim Paulson" "" "555-5678" "2001-05-16"
]
SQL "begin"
foreach [name address phone bd] data [
    SQL reduce [
        "insert into contacts values (?, ?, ?, ?)" name address phone bd
    ]
]
SQL "commit"
SQL ["DELETE from Contacts WHERE birthday = ?" "1967-10-10"]
results: SQL "select * from contacts"
probe results
disconnect db
halt


Rebol [ title: "MySql Example"]
do %mysql-protocol.r   ; http://softinnov.org/rebol/mysql.shtml
db: open mysql://root:%root--localhost/Contacts
; insert db {drop table Contacts} ; erase the old table if it exists
insert db {create table Contacts (
    name            varchar(100),
    address         text,
    phone           varchar(12),
    birthday        date
)}
insert db {INSERT into Contacts VALUES
    ('John Doe', '1 Street Lane', '555-9876', '1967-10-10'),
    ('John Smith', '123 Toleen Lane', '555-1234', '1972-02-01'),
    ('Paul Thompson', '234 Georgetown Pl.', '555-2345', '1972-02-01'),
    ('Jim Persee', '345 Portman Pike', '555-3456', '1929-07-02'),
    ('George Jones', '456 Topforge Court', '', '1989-12-23'),
    ('Tim Paulson', '', '555-5678', '2001-05-16')
}
insert db "DELETE from Contacts WHERE birthday = '1967-10-10'"
insert db "SELECT * from Contacts"
results: copy db
probe results
view layout [
    text-list 100x400 data results [
        string: rejoin [
            "NAME:      " value/1 newline
```

```
            "ADDRESS:   " value/2 newline
            "PHONE:     " value/3 newline
            "BIRTHDAY:  " value/4
        ]
        view/new layout [
            area string
        ]
    ]
]
close db
```

10.17 Graphics Examples

Displaying information in CRUD apps is often aided by the ability to use graphics. See http://business-programming.com for more information about how to create apps that use Rebol's powerful 'draw' dialect, as well as built in image handling capabilities.

```
REBOL [title: "Paint"]          ; draw with the mouse, save created images
view layout [
    s: area 500x400 white feel [
        engage: func [f a e] [
            if a = 'over [append s/effect/draw e/offset  show s]
            if a = 'up [append s/effect/draw 'line]
        ]
    ] effect [draw [line]]
    btn "Clear" [s/effect/draw: copy [line] show s]
    btn "Save" [save/png request-file/only/file %myimage.png to-image s]
]


REBOL [title: "Apply Effects to Downloaded Image Displayed in a GUI"]
view layout [
    pic: image load http://re-bol.com/palms.jpg
    text-list data [
        "Invert" "Grayscale" "Emboss" "Blur" "Sharpen" "Flip 1x1"
        "Rotate 90" "Tint 83" "Contrast 66" "Luma 150" "None"
    ][
        pic/effect: to-block value  show pic
    ]
]


REBOL [title: "2D Image Contort Draw Example"]
view center-face layout/tight [
    box 400x400 black effect [
        draw [image logo.gif 10x10 350x200 250x300 50x300]
    ]
]


REBOL [title: "Drawn Graphic, Cube With Gradient Color Fills"]
view layout [
    box 400x220 effect [
        draw [
            fill-pen 200.100.90
            polygon 20x40 200x20 380x40 200x80
            fill-pen 200.130.110
            polygon 20x40 200x80 200x200 20x100
            fill-pen 100.80.50
            polygon 200x80 380x40 380x100 200x200
        ]
        gradmul 180.180.210 60.60.90
    ]
]
```

```
REBOL [title: "r3D Lib Example"]   ; check out http://re-bol.com/i-rebot.r
do http://re-bol.com/r3d.r
Transx:   Transy:   Transz:   300.0
Lookatx:  Lookaty:  Lookatz:  100.0
do update: does [
    world: copy []
    append world reduce [
        reduce [cube-model (r3d-scale 100.0 150.0 125.0) red]
    ]
    camera: r3d-position-object
        reduce [Transx Transy Transz]
        reduce [Lookatx Lookaty Lookatz]
        [0.0 0.0 1.0]
    RenderTriangles: render world camera r3d-perspective 250.0 400x360
    probe RenderTriangles    ; This line can be removed
]
view layout [
    scrn: box 400x360 black effect [draw RenderTriangles]  ; basic draw
    across return
    slider 60x16 [Transx: (value * 600 - 300.0) update show scrn]
    slider 60x16 [Transy: (value * 600 - 300.0) update show scrn]
    slider 60x16 [Transz: (value * 600) update show scrn]
    slider 60x16 [Lookatx: (value * 400 - 200.0) update show scrn]
    slider 60x16 [Lookaty: (value * 400 - 200.0) update show scrn]
    slider 60x16 [Lookatz: (value * 200 ) update show scrn]
]


REBOL [title: "Little 3D Game, using native 2D graphic drawing routines"]
g: 12 i: 5 h: i * g j: negate h  x: y: z: w: sc: 0  v2: v1: 1  o: now
cube: [[h h j][h h h][h j j][h j h][j h j][j h h][j j j][j j h]]
view center-face layout/tight [
  f: box white 550x550 rate 15 feel [engage: func [f a e] [
     if a = 'time [
        b: copy []  x: x + 3  y: y + 3  repeat n 8 [
           if w > 500 [v1: 0]   if w < 50 [v1: 1]
           either v1 = 1 [w: w + 1] [w: w - 1]
           if j > (g * i * 1.4) [v2: 0]   if j < 1 [v2: 1]
           either v2 = 1 [h: h - 1] [h: h + 1]  j: negate h
           p: reduce pick cube n
           zx: p/1 * cosine z - (p/2 * sine z) - p/1
           zy: p/1 * sine z + (p/2 * cosine z) - p/2
           yx: (p/1 + zx * cosine y) - (p/3 * sine y) - p/1 - zx
           yz: (p/1 + zx * sine y) + (p/3 * cosine y) - p/3
           xy: (p/2 + zy * cosine x) - (p/3 + yz * sine x) - p/2 - zy
           append b as-pair (p/1 + yx + zx + w) (p/2 + zy + xy + w)
        ]
        f/effect: [draw [
           image logo.gif b/6 b/2 b/4 b/8
           image logo.gif b/6 b/5 b/1 b/2
           image logo.gif b/1 b/5 b/7 b/3
        ]] show f
        if now/time - o/time > :00:20 [alert join "Score: " sc q]
     ]
     if a = 'down [
        xblock: copy []  yblock: copy []  repeat n 8 [
            append xblock first pick b n  append yblock second pick b n
        ] if all [
            e/offset/1 >= first minimum-of xblock
            e/offset/1 <= first maximum-of xblock
            e/offset/2 >= first minimum-of yblock
            e/offset/2 <= first maximum-of yblock
        ][sc: sc + 1  if sc // 3 = 0 [f/rate: f/rate + 1]  show f]
     ]
  ]] at 200x0 text "Click the bouncing REBOLs!"
]
```

10.18 Sound Apps

Using sound in CRUD apps can be helpful. See http://business-programming.com for more information.

```
REBOL [title: "Wave File Jukebox"]  ; play sound files in selected folders
change-dir %/c/Windows/media
files: does [remove-each f copy read %. [%.wav <> suffix? f]]
view layout [
    t: text-list data files [attempt [
        insert s: open sound:// load value  wait s  close s
    ]]
    btn "Dir" [attempt [change-dir request-dir  t/data: files  show t]]
]


Rebol [title: "Chord Looper"]; backup music track, accompaniment generator
do load-thru http://re-bol.com/chords.r
check: does [if unset? snd/state/inBuffer [break] if not viewed? w [quit]]
view w: layout [
    h2 "Chords:"  a: area {bm bm g g d d em gb7}  across
    h2 "Speed:"   f: field 50 "3.3"
    btn "PLAY" [
        snd: open sound://
        forever [check  foreach s load a/text [
            check  insert snd do s  wait snd  wait (1 / load f/text)
        ]]
    ]
    btn "STOP" [close snd]
    btn "Save" [attempt [save request-file/only/save a/text]]
    btn "Load" [attempt [a/text: load request-file/only show a]]
]
```

10.19 Built-In Help and Reference Materials

```
REBOL [title: "Built-In Help and Reference Materials"]   ; compressed code
write %rebol-reference.r decompress #{
789C8554CB6EDB3010BCEB2BA6BA3846614836EC34151018688B3E809E9A3E0E
820230141DB3A14581A46C2945FFBD4B4AB6D536452F9248ED637676768DB8D3
0AB9934E890CF1A7707C2F541D17A88DAC1CE238929515C6CDC45E546EB6692A
8EF0C83710057249AFC475B5C035260FA243DE3BEEB42AE90FDD14641719B1D3
7B31138C6F2171B1E1CE66E0BAEE60F7FB64C3B870B38336A59D22675517F238
A9AB353259445B5567C823C0CA4781178BB45DA529B05DE2C7D70F6F70E33A25
2C2EDE7DF980832CEF85B3D3EC27993BD1BA9992D6617E95B6F3558A923906BA
358CBB907A2FCB99EDFD17210525212C1E11588F70A3CD0E8F19DD2BF1B7D79E
A946808C7863C14200E2E43109D50C1101A688437CD7B2427C530B2E9922C261
44AD2853C2941AD20C7EF1FAC440EC9BE06314F42CC8C535A64214CA8F3FB24E
370EDFBC53168F0B4E57E7828F90FBD87D8421C05B4FFDFFFC7DBBCE6ECC813A
B0A00EB00CCC0886E59CD8BD4C8790AF1AA91CA852B963F7BE2F5C49FEE0EB83
6277424D2911E3465B1B1511AF5C86B9E7CF0CE290A40C0F387813CB9A7C3D8D
C46AB87A464AF3825B936D4F2FAB6B5195209510C8B2A1CEE593604A16054626
89AE5417ECB8DED5DA9261DF98C07DED514A7B21AB52B46B6CE84D47C8E90CF3
22440960E981E70499F0F8CF6BBCA4286704939EA7D02CAA6FF4271F5AB75DA0
A11446C94A20FEEC6787240F3F3C4E93CA0836D1ED8F5C97222EA2C3302A7941
10B4D2C68E2ECEC4F5629586BA673BEBC46EA441E6E85CBB30AFBFCF174D0BA5
0DD2F0840E780F9045F1879F6B6A25FEE5D0E33AFF206FA27E5C3D45D346DE93
2E964B2F9F5E7E030E1B8F843D52E12A6DC97A50B1A64E1D90B7D9294D3F7CB4
24DAE3BEBA4D6E93B838E7BA4C432E1F125729EEFC5A8A5FF7249E72DADA8F7B
DAA684996748A323A9A1AA636D81C727A416624F369AD663D84FF314978BC135
082B7C062152707E120F4967F19474028440DF5E8A0338ED5D61FA8DA4FA7127
E35F4570D9F1BB050000} launch %rebol-reference.r
```

11. Learning More

See http://re-bol.com/rebol_quick_start.html, http://withoutwritingcode.com/ and http://musiclessonz.com/rebol_video_links.html for more gentle introductory understanding about Rebol.

When you're ready to learn everything in depth, read http://business-programming.com (that's an 800 page book with links to just about every other learning resource available to Rebol users).

Go to http://rebolforum.com to ask questions.