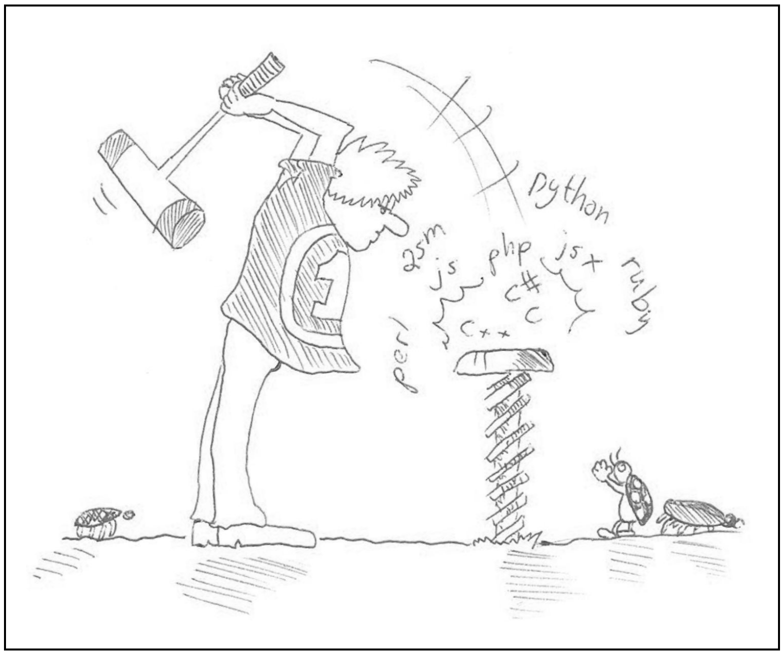
FriendUP Developer's Manual

Volume 1

"Draft", Rev. 5



Index

Index 2

Introduction 7

Why write a new operating system? 8

About undocumented code 8

Notes about this document 8

Authors 8

System layout 9

Security Model and components 9

System:Libraries/ 9

System:Modules/ 10

System:Devices/ 10

System:Devices/DOSDrivers/ 10

System:Documentation/ 11

System:Functions/ 11

System:Settings/ 11

System:Tools/ 11

System:Software/ 11

Friend DOS 13

Basic operation using Friend Shell 15

Friend Shell's DOS commands 16

cd 18

clear 18

flush 18

dir 18

list 18

type 19

set 19

echo 19

say 19

enter 19

leave 20

launch 20

status 20

break 20
kill 21
execute 21
engage 21
access 21
protect 22
assign 22
info 23
mount 23
unmount 23
mountlist 23
rename 24
makedir 24
copy 24
move 24
delete 24
wait 25
tinyurl 25
date 25
help 25
exit 25
Advanced shell scripting 25

The Friend Workspace 28
Friend Workspace core applications for development 28
Friend Shell 29
Friend Create 29
Calculator 30

Friend Libraries 31
system.library 31
Using devices 31
mount – mount user device 31
unmount – unmount user device 32
list – return list of mounted devices. 32
listsys – return list of available filesystems. 33
refresh – refresh FC device file structure (read settings from DB) 33
share – share device with other users across one server 34
Handling files 34
read – read a file from disk 34

write - writes to a file on disk 35
copy - copy a file to another 35
upload - uploads files to the Friend system 36
expose - create a public permanent link to a file 36
conceal - make a publicly available file private 36

Friend Modules 38
System.module 38
Administration commands 38
Calls available to users 42
Developing your own modules 43
Programming modules in PHP 43
The PHP database object 44
The PHP SQLDatabase class 44
The PHP Door class 45
Methods in the Door class 45
The PHP File class 46
Methods in the File class 46

Programming in the Friend Workspace 48
Friend Applications 48
The Config.conf file 48
The Application object 49
A very simple Friend application example 50
Communication between applications 51
Callbacks when messaging 51
Methods in the Application object 52
Callback functions in the Application object 52
Localization 53
Sharing application data between users 53
Shared Application Session class 54
SAS.invite(users, inviteMessage, callback) 54
SAS.remove(users, removeMessage, callback) 54
SAS.getUsers(callback) 54
SAS.send(event, username) 54
SAS.on(event, handler) 55
SAS.off(event) 55
SAS.close() 55
Event overview 55
Events for host mode 55

Events for client mode: 55
Application specific events 56
Shared Application Session example app: Whiteboard 56
Helper functions 57
Storage functions 57
Encoding and decoding of data 57
DOM helper functions 58
Localization functions 59
String manipulation functions 59
Dialog functions 59
The View class 60
Methods in the View class 61
The Widget class 62
Methods in the Widget class 63
The Screen class 63
Methods in the Screen class 64
File dialogs 65
The File Class 65
Friend paths 66
Methods in the File class 66
Public variables in the File class 67
The Door Class 67
The Module Class 68
Methods in the Module class 68
The Library Class 68
Methods in the Library class 69
Programming GUIs 69
How to write templates 69
A simple layout with a bottom bar 70
The horizontal tab layout 71
A double column layout 71
Triple column layout using nesting 72
Vertical layouts 72
Lists 73
A list of classes and their description 73
Mouse pointers 74
Absolutely positioned elements 75
Pulldown menus 76
A matter of scope 76

Tabs 77
Tree views 78
Directory views 78

Programming the Friend Core 80
Getting started with Friend Core and HTTP/S 80

DOS Structures 82
Example of a file structure 83
Example of a directory structure 83

Using DOS drivers 85
The Website DOS driver 85
The Server DOS driver 89

Terminology (alphabetized) 90

Introduction

Welcome to this Developer's Manual for the Friend Unifying Platform. We have worked hard to create a platform that may inspire you and hopefully encourage you to take your computer(s) and the internet to the next level. At first sight, Friend may look like any other desktop environment and web server. But once you start digging deeper, you will find a new and unique architecture that simplifies how you connect to and process data over a network.

Friend inherits from many operating systems and servers that have been in use over the last 40 years. On the server side, you will find aspects of Apache and Nginx. On the OS side, you will find Tripos, Amiga OS and Linux. On the UI side, you may find aspects that remind you of Android and Tizen. You will realize that Friend is a departure from the standard Unix philosophy. And that is so - Friend follows the path that Tripos ventured - a simplified and reorganized take on Unix that works much better for users and systems with multiple root devices. We believe you will find that it is very well suited for user interfaces and for abstracting the internet as a sea of resource islands to be utilized and disposed of.

In this document, we will cover the entire system from A to Z. We will explain how the system is designed. We will explain each system component. Then we will go into the various programming languages you can use. We will cover the classes and functions you might use. And we will give lots of examples. Friend, essentially being an OS, has components dealing with everything from files to devices. Because of this, it might be a bit untraditional as an online, web based environment, compared to what you have been using in the past.

The most untraditional aspect of Friend is that it is implemented using a "operating system template". What we mean by that, is that Friend implements the components of an OS; drivers, libraries and resource management functionality. The rationale is that you need these structures to write complete computer programs (and you always did). This is why you will be able to create very powerful and easy to distribute applications in Friend.

By historical tradition, an operating system consists of four distinct parts of a dualistic whole. On the bottom layer, you have the **kernel**. On top of that, you have the **kernel shell**. This allows for communication between the kernel and the outside world. Then you have the **desktop shell**, which expands into the **graphical user interface**. Each part plays a specific role in the system. The kernel abstracts raw data and manages internal processes and resources. The kernel shell creates a low level user interface that exposes kernel functionality to a user. The desktop shell abstracts the kernel shell into objects that can be represented graphically. The graphical user interface draws the graphical representations of the desktop shell and implements an interactive toolkit to manipulate the operating system.

Why write a new operating system?

First, it is rare in any field, let alone computer operating systems, to have the luxury to write a new OS from scratch, and hope to replace what has been in wide use for many years, or decades. While many users crave a clean fresh start, in reality, so much would break, or have to be re-integrated with existing hardware and infrastructure, that the re-deployment effort and time-span would discourage many from even attempting to try it.

A critical architectural decision was made to strategically and efficiently "leverage" the stable facilities of the main existing operating systems, kernels, and browsers, and integrate our advanced “co-kernel”, Friend Core, or meta-layer, if you will. We place this layer above these time-tested, established systems. This way we can utilize them to deliver rapid and broad deployment across most of all existing computing platforms. Immediately upon launch, this meta OS runs everywhere on nearly every platform, both legacy, and new devices just being introduced!

We believe that the changes we have made provide a more natural embrace to web servers and standards, modern extensions to the computing environment such as voice input and output, VR and AR, IoT, AI, as well as thorough utilization of the cloud. The FriendUP platform truly and elegantly provides the individual user with the access and power of a planetary computer! We call that a *macro computer.*

About undocumented code

This document covers the API v1 specification. Undocumented function calls and classes that may be found in the FriendUP source code, may become obsolete and deprecated without warning. We urge you not to utilize these functions and classes, as they may render your application unusable in the next update of the system.

Notes about this document

This manual is a work in progress. Consider it a preview of what is to come in version 1.0. Some of what is documented may have problems. In some cases features may simply not work. But do not see this as a warning. Friend is ready for serious third party development. So even though this is a preview, it will still let you unleash your creative potential and learn about how to develop applications in Friend. Have fun!

Authors

Hogne Titlestad, Thomas Wollburg, Francois Lionet, Paul Lassa, Paweł Stefański.

System layout

Friend has an operating system template that abstracts information and data structures in system components of various kinds, depending on kind. Using these components, FriendUP allows a developer to build rich applications that can access any type of information or service over a network.

Security Model and components

FriendUP is designed from the ground up for use in the modern enterprise. As such, it offers the full spectrum of industry standard security features and components to fit seamlessly into existing IT infrastructures. It is designed to protect and preserve critical data, properties, and communications of commercial entities and individual users.

Friend Core can run using SSL/TLS for HTTP and websocket connections. All production level environments should use SSL at all times. Friend Core requires authentication to access any core logic. Additionally, it is mindful of user levels and permission setups.

By using sandboxed iframes and worker threads, Friend puts up a boundary between each Javascript application and the core system. This way, each application is forced to adhere to its security setup, enforced from the time it is installed in the Friend Workspace. Friend uses security subdomains to sandbox Javascript applications, and string based application-to-application messaging for accessing the Friend API. This way, no application can share any Javascript memory structure, preventing offensive applications from escaping their sandbox.

Friend supports third party identification providers. More info about that can be found in our Administration guide.

System:Libraries/

System libraries are collections of functions that are executed inside of Friend Core, the Friend kernel. This allows them to execute with optimal speed without having to initialize runtime environments or reprocess data that is already prepared in the core.

The system.library can be reached using many programming languages, like PHP and Javascript. It allows you as a developer to take advantage of the many fast and powerful functionalities inside of Friend Core in your own applications.

Other libraries will be documented in a revised version of this manual.

System:Modules/

System modules abstract functionality that executes outside of the Friend Core. They can be written in several languages and will execute on the server. By using a module, Friend Core can be extended with any scripting language, as long as it returns data formatted in adherence with the Friend Core module specification.

Friend Core comes with several modules. But the two modules you will be using the most are the *system module* and the *files module*. These cover the most important Friend functionality that you would use in a Friend application.

System:Devices/

The devices directory of the System volume is designated for device drivers like DOS drivers, printers, network nodes and hardware devices.

System:Devices/DOSDrivers/

DOS drivers are virtual filesystem drivers for Friend Core. A Friend filesystem device is handled by a DOS driver. The driver handles DOS command calls and paths directly, and returns data in adherence with the Friend Core DOS driver specification.

In Friend, a disk volume may be anything. We have designed the data structures that are found on these disks in a way that they can abstract any kind of data. Friend supports not only javascript executables on these disks, but also libraries that relay remote server functionality (please read about the Website DOS driver elsewhere in this document). This way, a disk may contain several independent applications that are distributed from remote, trusted infrastructures like cloud servers or purpose built proprietary servers.

The DOS driver is one of the most powerful features in Friend. These provide middleware functionality and package it in a coherent and easy to use interface for the user. Our open source package contains a couple of DOS drivers to review and to be taken as blueprint for creating your own drivers. The driver may connect to any structured or unstructured data source, from filesystems via legacy databases to whole applications that can be laid out as metadata-rich files and directories.

To illustrate the usefulness of DOS drivers, an example use case would be creating a driver for your warehouse database. Then easily publish products stored there by drag and drop to a Wordpress DOS driver that connects to your website and online store.

System:Documentation/

The documentation directory contains, among other things, this documentation. It is the one stop place where you can reach the documentation needed to learn and understand your Friend system on all levels. When you expand your system with additional software and upgrades, you will see an increase of documents and document directories.

Request: If you are unable to find coverage of a particular topic, please submit a request to: ***developer@friendos.com***, and our team will try to address the gap!

System:Functions/

The functions directory contains Friend DOS functions that can be used to manipulate your Friend environment. This is also the directory where you can extend your Friend system with new commands that you have either written yourself, or third party creations that you have installed.

Friend DOS functions are listed and more fully described in the **Friend DOS** section.

System:Settings/

This directory contains your system settings. Here you will find the applications that lets you modify your Friend Workspace. Administrators gets some extra applications to set up user accounts and do other administrative tasks.

System settings apps are listed and described in the Friend **User's Guide.**

System:Tools/

Contains bundled system tools. Here you will find applications that let you monitor the active state of your system. As you expand and add to the software available in your Friend system, additional tools may appear here. System tools apps are listed and described in the Friend **User's Guide.**

System:Software/

Contains a categorized directory tree of all your available software. When you install new software in Friend, this software can be found in the System:Software/ directory.

This directory is similar to the *Program Files* or *Applications* directories on other systems.

Note that while the software appears to be installed in this local directory, it may in fact reside on some other server (with Friend Core, or on your uncle's, neighbor's, sister's hairdresser's computer/server. But, it now has been ***"enabled"*** to be run in your Workspace. Friend enables decentralization of software sources. But each user account keeps track and manages its own software authorizations and application permission settings.

Friend DOS

The Friend Disk Operating System is the environment where a user can access the kernel shell using a ***command line interface*** (**CLI**). It is available using HTTP, WebSockets, SSH and the Friend Workspace application Friend Shell.

The Friend DOS syntax is similar to Unix shell CLI conventions, but closer to the DOS implementation found in the AmigaOS and Tripos operating systems. The Unix file and directory layout is modelled on a ***single hierarchical root*** ("/"), while Tripos allowed for ***multiple roots***, which better fits our current times where we are used to working with a plethora of network mount points.

One of the design decisions taken by the FriendUP team when first creating the foundational layers of the system was to utilize DOS as a unifying protocol between the Friend subsystems. DOS can manage all data resources and deliver predictable results. And through FriendUP, DOS also serves as the ultimate high-performance, low-overhead, **system-to-system API,** or data and resource conduit to and from the major established OS platforms. (Networking protocols, the basis of machine interconnect of the internet, are similarly useful, but impose extra overhead and complexity for unreliable and latency-varying connections, which reduce performance.)

Once DOS connectors are enabled, any system or platform connected to Friend DOS now becomes part of the larger meta-system. In fact, a Linux app can be made to co-operate or co-function with another Windows app, and each may not even be aware who or what they are connected to. One can be writing data to a file (a normal function on most operating systems), While the other is reading / consuming that same data from the shared file or device, and performing secondary operations on that data. For example, a modern web application running on Linux can exchange/receive data to/from a legacy database or spreadsheet running on a Windows desktop PC.

Friend DOS is also close to the spoken language, which offers many advantages like speech to text processing directly to computer logic. Thus, most DOS CLI actions make sense from a grammatical standpoint, and programs or devices (nouns) can take actions (verbs) on other objects or devices. The user can speak DOS function commands or sequences to the platform, and Application operations can be triggered, and then status or notifications can be spoken back as a result. A simple example of this could be verbally controlling lighting in a house or workspace, or commanding to stream a downloaded video or music file to a secondary display device or TV. While many may still prefer to type commands, Voice-controlled UI, and its connections to AI (Siri, Alexa/Echo, Google Assistant, Watson, etc.), will continue to grow in usage and popularity in the coming years.

For those who are not familiar with how a DOS CLI works, the basic operation is explained here. A CLI is essentially a textual interface to the operating system. It allows you to write simple commands with their arguments which are then parsed to give you a result. To be able to fully understand how it works, we need to explain how the **file structure** works.

A file structure represents data on a disk using a spatial metaphor, as a hierarchical tree, or as Friend DOS permits, many hierarchical trees. It presents you with a container, a disk, wherein you can have directories (subcontainers) and files (blocks of binary data represented by a file name). In a CLI, you have a shell prompt (or path), which represents your current reference location - where in the structure you are right now. The prompt says something like:

```
1. System:> _
```

This tells you that you are located in the **System:** disk - equivalently that your "***current directory***" is "System:". The number "1." indicates to you that this is shell process number one. When opening up new CLIs, each consecutive session is given a new number. When changing to a new location, you use the **"change directory"** command, aptly named ***cd.***

```
1. System:> cd Modules/
1. System:Modules/> _
```

And now the prompt changes, to reflect your new position (or path) in the tree. The rightmost name 'Modules' is now your current directory location, and the rest of the path, to its left, represents the outer or upper container(s) of your current directory. Thus, the "Modules" directory is a sub-container "within" the "System:" directory or container (or disk volume).

Note: The very first, leftmost Name in your prompt string is called the ***device or volume root***, and it is always followed by a colon character, "**:**". After the ***Root:Directory-name***, subsequent ***subdirectories*** in the path are separated with a "**/**" (forward slash) character, as in "Root:Directory/Sub-directory/Sub-sub-directory/", with a "**/**" tacked onto the final, ***rightmost directory-name***, in this example, "Sub-sub-directory/". Then finally as padding characters, there is a "**>** " (greater-than character and space), representing the end of the prompt string, where your next typed command will appear. Also note that you refer to files with just their ***filename***, while you refer to directories and Sub-directories with their ***Directory-name***, appended with a '**/**' (forward slash), as in "dir-name/". Therefore a directory can contain both a file named, "Test", and a subdirectory named "Test/". Your DOS command will reference the **file** "Test" instead of the **directory** "Test/" if you have not added the trailing "**/**".

Once you get used to thinking about the CLI as a spatial interface to a completely abstract world of binary data and other digital structures, it can help you to uncomplicate your computer system and really make you understand it in a simple and elegant way (like a house, with floors, rooms and compartments).

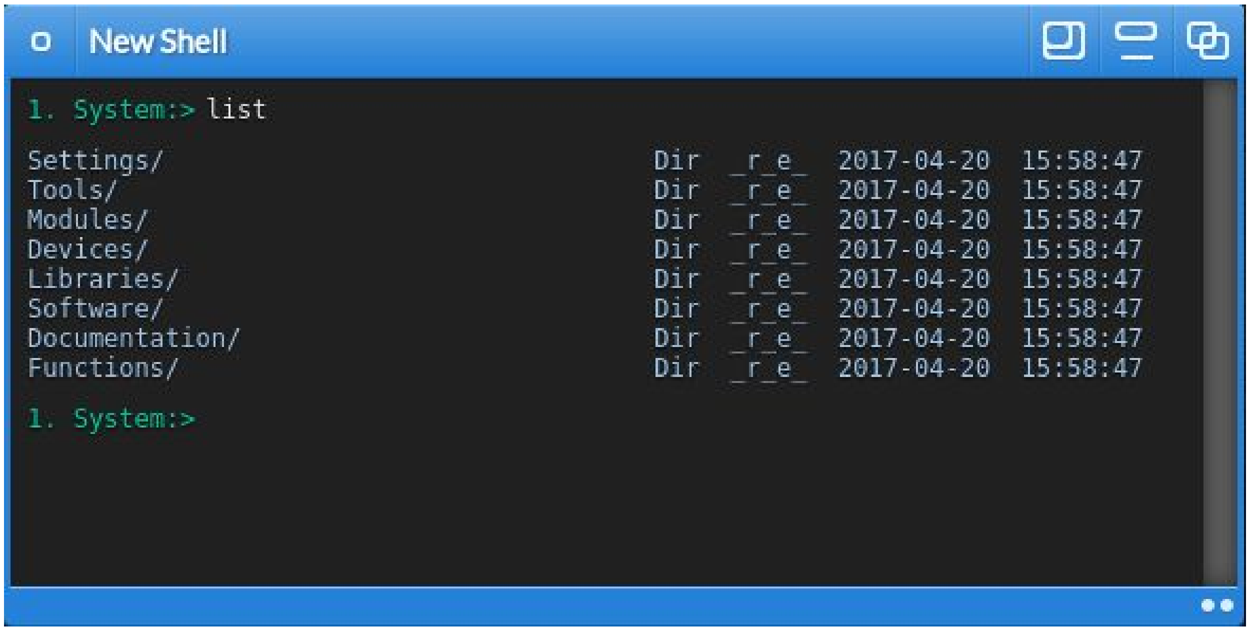
It is possible to have a fully usable Friend system even though you are only accessing it using a CLI. The GUI in Friend is a mere storefront to functionality that is running behind the scenes, as it were, no matter which user interface you apply to it.

Basic operation using Friend Shell

When opening up the Friend Shell, you are presented with a standard CLI prompt. Unless you have modified the default Shell settings, you will be greeted by the System: volume when opening the Shell. To see the files and directories that are located in this volume, type the following:

```
4. System:> list
```

Then press enter. This will give you a listing of all the directories and files located at your current path (which is System:).



```
1. System:> list

Settings/                              Dir  _r_e_  2017-04-20  15:58:47
Tools/                                 Dir  _r_e_  2017-04-20  15:58:47
Modules/                               Dir  _r_e_  2017-04-20  15:58:47
Devices/                               Dir  _r_e_  2017-04-20  15:58:47
Libraries/                             Dir  _r_e_  2017-04-20  15:58:47
Software/                              Dir  _r_e_  2017-04-20  15:58:47
Documentation/                         Dir  _r_e_  2017-04-20  15:58:47
Functions/                             Dir  _r_e_  2017-04-20  15:58:47

1. System:>
```

The list command lists one file element per line, along with its file-***type*** (Dir=directory, or other file-content type), its ***combined access permissions*** *string,* as well as its ***modification*** date and time. The 5-character access permissions string is of the form, “**`a/r/w/e/d`**”, where a=archive, r=read, w=write, e=execute and d=delete.

When a character is not present in the access permissions string, a dash "-" appears in its position, and this indicates that the file does not have access permission for the corresponding operation. For example, if there is an "-" in the "w" position, that means that the file may not be written to (or overwritten). Similarly, if there is a "-" in the "d" position, the file may not be deleted or removed, unless that access flag is changed to "d".

The file ***modification string*** is in the format of **`year-month-date hour:minute:second.`**

See also the **access** and **protect** commands below.

From issuing the **list** command above, you can see the main directories in the System: volume listed out. You will notice that only directories are listed here, indicated by the **"Dir"** label in the type column. After it follows **"-r-e-"**, which means that the directories are read only, and executable. After all, System: is a read only ***virtual file system.***

Friend DOS scripting is a bit different from many other scripting languages. We thought long and hard on risking to depart from what people are familiar with in the Unix world (or Windows world for that matter), but we came to the conclusion that - since FriendUP is so different in how it connects to resources, we ought to reflect this in our DOS implementation.

At a basic level, you can do normal things like:

```
4. System:> cd Home:
4. Home:> dir
4. Home:> output Readme.txt
```

This would enter the Home: volume, give you a directory listing and output the contents of a file with the name Readme.txt.

Friend Shell can also understand inferred queries, like:

```
4. System:Modules/> /
4: System:> Home:
4: Home:>
```

Here, you start out in System:Modules/ and go to the parent directory by using a "**/**" (forward slash). Then you switch to the Home: volume by evoking its name (including the colon). The shell has a few of these convenient shortcuts. More on them later.

Friend Shell's DOS commands

Friend command line options follow the below template:

```
command argument variable=value
```

Variables are specified by adding a value after the equals(=) sign. **Arguments** are valueless command line options that instruct or modify how the command operates.

Sometimes, you want to add in a ***to*** argument to separate logic arguments in a command line query, like this:

```
1. System:> copy Home:directory/ to Otherdisk:destination/
```

These arguments are only there to help you remember how to write the CLI command queries. You could skip them, and simply do:

```
1. System:> copy Home:directory/ Otherdisk:destination/
```

Additionally, you may want to add the recursive argument ***all*** to your command line to make commands like delete and copy perform ***recursively***, like this:

```
1. System:> copy all Home:directory/ to Otherdisk:destination/
```

Let's take a look at the ***most common shell commands*** and how to use them. If you know the following commands, you should be quite proficient in the Friend shell:

• access • assign • break • cd • clear • cli • copy • date • delete • dir • Echo • engage • enter • execute • exit • flush • Help	• info • kill • launch • leave • list • makedir • mount • mountlist • move • protect • rename • say • set • status • type • unmount • wait

cd

The command *cd* is short for “***change directory***”. Type it to move your prompt into another path or *mount point* on your system. You can also just name the directory instead of typing *cd*, as a *shorthand* way to achieve a directory change. To get to a ***parent directory***, you type: “**cd /**” or just “**/**”. If you want to get to the root of a directory, for example from Home:Documents/Drafts/ to Home:, you type “**cd :**” or just “**:**”. Examples below:

```
4. System:> cd Home:
4. Home:> Documents/
4. Home:Documents/> /
4. Home:> cd Documents/Drafts/
4. Home:Documents/Drafts/> :
4. Home:> System:
4. System:>
```

clear

The *clear* command cleans up the shell log in your current shell view window and positions the prompt and cursor at the top left of the window.

flush

Flushes all variables (removes them from memory) from the current shell session. This operation can not be undone. This command can be good to have around when running sequential scripts.

```
4. System:> set a 5
4. System:> echo $a
5
4. System:> flush
4. System:> echo $a
$a
```

dir

The *dir* command generates a simple directory listing of your current directory path. It organizes directories first, then normal files. It does not list specific information about each file, only the filenames. The ***ls*** command is an alias to *dir*.

```
4. System:> dir

Settings/    Devices/      Documentation/
Tools/       Libraries/    Functions/
Modules/     Software/
```

list

The *list* command generates a more detailed directory listing of your current directory path. It organizes directories first, then normal files. It lists file ***size***, ***combined permissions*** for each file, as well as ***modification date***.

```
4. Home:Documents/> list
Recipes.odt            206kb      -rwed       2017-04-26 11:00:01
Jokes.rtf              110kb      -rwed       2017-04-26 12:11:02
Thesis.odt               3mb      -rwed       2017-03-20 11:10:00
```

type

The *type* command outputs the contents of a file to the shell output buffer. This is similar to the unix/linux 'more' or 'less' commands.

```
4. Home:> type test.txt
Welcome to test
--------------
This is a text document
…
```

set

Sets a shell variable to a value. This is similar to the unix/linux "setenv" command.

```
4. System:> set a 5
4. System:> echo $a
5
```

echo

The echo command outputs some text to the shell output buffer, which by default, displays character output in the shell view window. See also **input/output stream redirection** below.

```
4. System:> echo "Hello world!"
Hello world!
4. System:> echo "The number A is $a"
The number A is 5
```

say

The say command is similar to the echo command, only that it uses the computer voice registered with your Friend system to speak the text within quotes. As the example below shows, this can also include evaluated or substituted values of shell variables, as the ***value*** of the shell variable '**a**' is substituted in for **$a** at the end of the quoted string.

```
4. System:> say "The number A is $a"
```

enter

The enter command changes directory to the **Functions/** directory in the specified Dormant disk volume. In this command, the trailing colon "**:**" of the *volume-name* need not be specified, it is assumed.

```
4. Home:> enter System
4. System:Functions/>
```

leave

The leave command reverts the prompt back to the previous path before **enter** was issued.

```
4. System:Functions/> leave
4. Home:>
```

launch

Executes a Friend application detached from the shell. Can take arguments. The example below shows just starting the application **Friend Create**, and then starting the same application with a file as argument 1. The launch command is similar to executing a Unix/Linux command with a trailing '**&**' (ampersand), which means to spawn a new stand-alone process that is not a child of the current shell process.

```
4. Home:> launch FriendCreate
4. Home:> launch FriendCreate Home:Projects/test.jsx
```

status

Generates a list of the Friend applications or processes that are running. This includes both foreground and background (detached) processes. The processes are listed in ascending order by **task id**. FriendCreate has task id of "1", Author has task id of "2", and FriendShell has task id of "7". Note that the command prompt, "**4. System:>** ", has shell id of "4", not to be confused with task ids.

```
4. System:> status
1. FriendCreate
2. Author
7. FriendShell
```

break

Shuts down or terminates a Friend application or process by **task id**. In the example below, Friend Create is shut down and removed from running applications. See also command "**kill**".

```
4. System:> break 1
```

kill

Kills or terminates a Friend application by name. See also command "**break**".

```
4. System:> kill FriendCreate
```

execute

Runs a Friend DOS script. The output of the script is directed to the shell's standard output buffer, the shell display window. This output can be suppressed, or redirected to a file or another process or application. See **input/output stream redirection** below.

```
4. System:> execute Home:myscript.run
Welcome to this script!
We are now counting from 1 to 10:
1..
2..
...
```

Friend DOS scripts are human-readable text files that contain one or more command line strings. They may also use more advanced scripting syntax. **Advanced scripting** is in the next sub chapter.

engage

The engage command enters into, or attaches to, a running Friend application. The input and output is from then on managed by the Friend application, and no longer the Friend Shell. This is useful if you want to use Friend Shell for debugging or to log into external systems. Example:

```
4. System:> cd Home:Programs/
4. System:Programs/> list
MyExampleProgram.jsx           32kb      -rwed        2017-04-26 14:17:04
4. System:Programs/> launch MyExampleProgram.jsx
4. System:Programs/> status
1. MyExampleProgram
4. System:Programs/> engage with MyExampleProgram
> Welcome to My Example Program
>
> What do you want to do? _
```

access

The *access* command provides you with a list of the ***access privileges*** that are set on a file or a directory. Usage:

```
4. Home:> access myfile.txt
The access privileges of Home:myfile.txt is:
user: -rwed        group: -r-ed        others: -----
combined: -rwed
```

The ***combined privileges*** match what is listed for each directory or file when issuing the "**list**" command. See also the **protect** command below.

protect

To change the ***access privileges*** of a file or directory, you use the protect command. The syntax is simply to specify the file you want to protect, and then add the privileges for each **access category; user, group and others**. You do not need to set the privileges for all access categories, but if you would need to, then the syntax would be as below:

```
4. Home:> protect myfile.txt user=rwd group=r others=-
Permissions were set.
```

This is the same as setting user=rwd:. For group and others, you need to specify.

```
4. Home:> protect myfile.txt rwd
Permissions were set.
```

See also the **access** and **list** commands above.

assign

The *assign* command is very powerful and must be used with care. It creates ***new virtual disk drives*** based on directories of other existing drives. In other words, it assigns or combines multiple different directory paths into new virtual disk drives that can be seen as ***merged file resources.***

```
4. System:> assign Home:Wallpaper/ to Imagery:
4. System:> list Imagery:
Balloons.jpg            36kb      -rwed        2017-04-26 14:17:10
Dark_Cave.png          245kb      -rw-d        2017-03-20 12:10:01
4. System:> assign Storage:Images/ to Imagery: add
Path Storage:Images/ added to Imagery:.
4. System:> list Imagery:
Amigos.gif              16kb      -rwed        2017-04-26 12:10:07
Balloons.jpg            36kb      -rwed        2017-04-26 14:17:10
Dark_Cave.png          245kb      -rw-d        2017-03-20 12:10:01
4. System:> assign Store:Images/ remove from Imagery:
Path Store:Images/ removed from Imagery:.
4. System:> list Imagery:
Balloons.jpg            36kb      -rwed        2017-04-26 14:17:10
```

```
Dark_Cave.png          245kb      -rw-d        2017-03-20 12:10:01
```

Note the use of one or two arguments in each of the assign commands above. As noted earlier, the "**to**" and "**from'**" arguments are optionally used to clarify the command relationship between two directories named in the command. Also, an "**add**" argument is issued in the assign command to specify that the source specified directory, "`Storage:Images/`", rather than becoming **the new** "Imagery:" path assignment, is instead **added** to the existing "Imagery:" assignment, giving it now two component directory paths. This can be done multiple times, up to the assign path limit. At that point, one of the *previously* **added** directory paths would need to be **removed** first, before another could be **added**.

Assign drives behave pretty much like normal disks. They can be mounted and unmounted and they can have visibility or be hidden. See also the **mount** and **unmount** commands below.

info

Gets file information for a file. Shows all the relevant file attributes in a list.

mount

Mounts a drive that is then ***available*** in the **mountlist**. When a drive is mounted, it will show up on the Workspace desktop if it is ***visible***. If it is mounted, but ***not visible***, then it will not appear in the Workspace desktop, but it can still be accessed per its access permissions by Friend applications and commands executed in the CLI DOS shell. See also the **unmount** and **mountlist** commands.

unmount

Unmounts a drive that is currently ***mounted***. When unmounted, the disk will be removed from the system and is no longer available for any disk operation. If the mounted drive had been visible on the Workspace desktop, it will now be ***removed from view***. See also the **mount** and **mountlist** commands.

mountlist

Produces a list of ***available*** or ***unavailable*** disks registered with the Friend system. When the command is executed without arguments, it will produce a list of the mounted disk volumes. If it is executed with the argument ***unmounted***, then it will produce a list of the unmounted disk volumes.

```
4. System:> mountlist unmounted
Volumes:                    Type:                          Visible:
```

```
Test:                          Assign                                    no

Found 1 unmounted disk(s) in mountlist.
```

rename

Renames a file on disk. Works on both directories and files. The rename command is similar to the unix/linux **mv** (or move) command.

```
4. System:> rename file.txt to document.txt
```

If a directory is renamed after is has been added to a ***virtual disk drive*** using the **assign** command, it will stop being part of this volume.

makedir

Creates a new directory with a name given.

```
4. System:> makedir Mypath/
```

The makedir command is similar to the unix/linux **mkdir** command. See also **delete** command below, for deleting or removing a directory.

copy

Copies a file or directory to a ***destination path***. Can be recursive with the ***all*** argument, which is optional. By default, it is not recursive, and only copies the first level of files.

```
4. System:> copy all Home:Documents/ to Storage:
```

The copy command is similar to the unix/linux **cp** command. See also **rename** and **move** commands.

move

This command is the same as **copy**, only that it ***deletes the source files*** after the copy is completed. Use it with care.

delete

This command deletes a file or directory. With the ***all*** argument, which is optional, it deletes recursively. By default, it is not recursive, and only deletes the first level of files.

```
4. System:> delete all Home:Documents/
```

The delete command is similar to the unix/linux **rm and rmdir (remove and remove directory)** commands.

wait

Wait (or pause/delay) for x amount of ***milliseconds***.

```
4. System:> wait 5000
```

This DOS shell command uses the ***host system timer*** to introduce a measured delay between this command and any subsequent command.

tinyurl

Create an url that is synonymous with another. Only supports links inside your Friend system.

```
4. System:> tinyurl https://mycore.com:6502/webclient/index.html
```

The system will return with a hash that you can use. If you put it after your domain name, it will display the contents of the original link. Example:

```
https://mycore.com:6502/A549AB30/
```

date

Date outputs the current date and time.

help

Help gives a list of all the commands that are available in the shell. If invoked with a shell command as the second argument, it outputs a short description of how to use the shell command.

exit

Exits and terminates the shell session, freeing up the shell id to be used by a another shell session later.

friendnetwork

fnet (alias)

The 'friendnetwork' command and its shorter alias 'fnet' let you share or access shared Shell sessions on other computers connected to the network of Friend machines and servers.

You can very easily give access to your machine or a limited portion of it by hosting a FriendNetwork host shell. Other users will be able to connect to your machine in a transparent way, the host shell remaining unaffected.

friendnetwork host hostName [guestPassword]

Opens a host session in the current shell.

- **hostName**: the name you want to be broadcasted on the network. For example myShell, "Hogne's den", 'This is my shell' (remember to use quotes if you have spaces in the host name). Your host name will appear to other users as 'hostName@yourUserName'
- **guestPassword**: defines a guest restricted access password to your Shell.
  When another user wants to connect to your hosted Shell session, he is asked to enter a password.
  If he enters the **same password** as the one you have used to start Friend, he will have access to all the directories and all the commands of the Shell, as if was using your local computer. This option allows you to use a Friend machine as a SSH host for your personal use. Warning: external users will be able to delete files or directories, launch applications and batches, so be careful not to give your password to untrusted users.
  If you specify a guest password in the 'friendnetwork host' command, the externals users that connect with the **specified guest password** will only have a restricted access to your machine :
  - they will only have access to the directory you were when typing the 'friendnetwork host' command and its subdirectories. For example, if you start your host Shell session in 'Home:Documents/' a 'cd System:' command will report an error.
  - they will not be able to delete files or directories.
  - they will not be able to launch applications or run batches.
  Please note that even if you have specified a guest password, you can still access your Shell host using your main password, with full access.

friendnetwork list

Lists all the available hosts on the network. Each host is listed in the form of : hostName@hostUserName. Use this command to discover if interesting hosts are open, and use their names to connect to them with the 'friendnetwork connect' command.

friendnetwork connect 'hostName[@hostUserName]'

Connects your Shell to a host.

- **hostName**: the name of the host you want to connect to. If this name is duplicated in the list of host available, it will connect to the first one.
- **@hostUserName**: by adding the host user name to the address, you will be sure to connect to the correct host.

Example:
friendnetwork connect arthur_shell : will connect to the first 'arthur_shell' in the list of hosts.
friendnetwork connect arthur_shell@arthur : will connect only to the host session hosted on arthur's Friend machine.
You have to enter a password to establish the connection. It can be (for your own use) your main Friend password, which will give your full control of the distant machine. If the host has defined a guest password and if you use it, you will only have a restricted access (see above).
Once you are connected, all the commands you type in your local Shell are re-routed to the host Shell, executed on the host machine and the result of the command is displayed on your Shell. This operation is totally transparent for the host, yet he will be noticed when you connect.

friendnetwork disconnect

Disconnects you from a host, if you were previously connected. The current path of the shell is restored to the one before the 'connect' command was typed, and all the commands you type later will have a local effect.

friendnetwork dispose hostName

Removes a hosting session from the network. All existing connected users will be disconnected, and your name will no longer appear in the list of hosts.

- hostName: the name you have used in the 'friendnetwork host' command.

friendnetwork password hostName newGuestPassword

Defines a new guest password for the given host session, and replaces the previous one. Guests that are already connected with the previous password will not be affected.

- hostName: the name of the hosting session
- newGuestPassword: the new password to use

friendnetwork status.

Returns a list of opened host and client session on the current Friend machine

Advanced shell scripting

Friend DOS accepts not only simple commands, but arguments, loops and jumps like any other scriptable interface. Let's go through some advanced shell scripting, starting with something relatively easy and then progressing to more complex scripts.

```
4. System:> Home:
4. Home:> repeat 5 times: output Readme.txt
```

This would output the contents of the Readme.txt file five times.

But we can do still more.

In FriendUP, applications have their own ***filesystems***. So here, we can use our **Friend Create** programmer's editor. We'll use it dormantly - using our **Dormant** technology (for more on Dormant and Friend Create, see their own chapters).

Written in the easiest shorthand way:

```
4. Home:> enter FriendCreate
4. FriendCreate:Functions/> LoadFile Home:Readme.txt
4. FriendCreate:Functions/> repeat 5 times n: ReadLine n
4. FriendCreate:Functions/> leave
4. Home:> _
```

Written in a way that programmers can comprehend (as viewed in a .run script file):

```
cd FriendCreate:Functions/
LoadFile Home:Readme.txt
repeat 5 times n:
       ReadLine n
leave
```

Writing this logic in a programming language like Javascript is more complicated than doing it in Friend Script. Nevertheless, an advanced programmer would always prefer latter way, because it gives more power and precision. On the other hand, Friend Script can also be accessed in Javascript, giving you the best of both worlds:

```
var ShellInstance = new Shell(); // Create a new shell session
ShellInstance.setOutput( 'console' ); // Sets the output of script to js console
ShellInstance.onReady = function()
```

```
    {
            // The semi colon in the inline script is instead of newlines...
            ShellInstance.execute( "\
            enter FriendCreate;\
            LoadFile Home:Readme.txt;\
            repeat 5 times n: ReadLine n;\
            leave\
            " );
    }
```

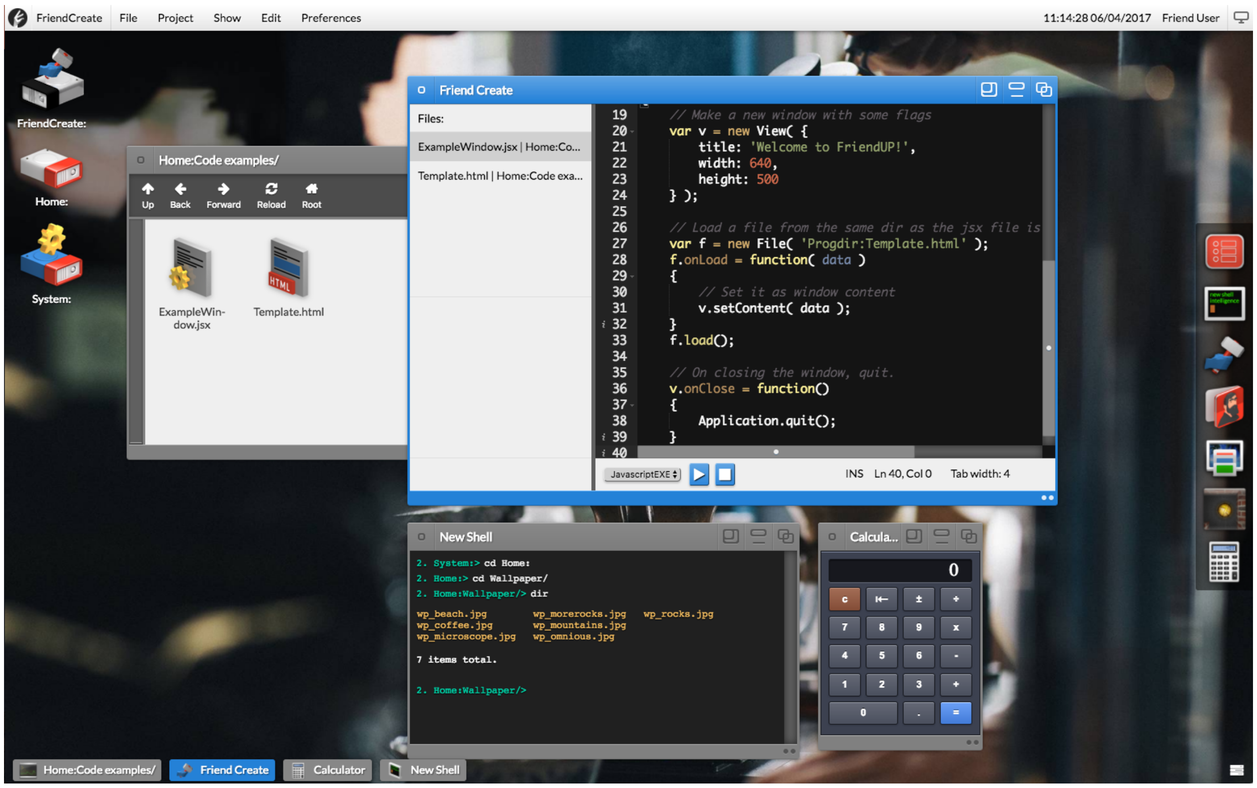
An important observation here, is that Friend Script is synchronous, whereas a language like Javascript is asynchronous. This means that in Javascript, you must resort to callback functions that are executed once certain events have fired. In Friend Script, the prompt waits until one operation is completed before going to the next step. This has its disadvantages in some cases, but Friend Script is better suited for linear tasks, and as such, being synchronous is overall a benefit to the programmer.

The Friend Workspace

The client side part of Friend is the Friend Workspace. It is written using HTML, CSS and Javascript. This makes it available on any platform and operating system as all meaningful operating systems come with a browser.

The Workspace provides direct access to the Friend Core server and comes with user, file, access and window management. It has a file manager and several default applications like Friend Create, our programmers editor, and Friend Shell, our command line interface.

The Friend Workspace provides a responsive desktop with view/windows, a dock, widgets, a global menu system and a mount list over available file systems. It also always gives access to the System volume that provides access to software, settings, tools and documentation.



```
    // Make a new window with some flags
    var v = new View( {
        title: 'Welcome to FriendUP!',
        width: 640,
        height: 500
    } );

    // Load a file from the same dir as the jsx file is
    var f = new File( 'Progdir:Template.html' );
    f.onLoad = function( data )
    {
        // Set it as window content
        v.setContent( data );
    }
    f.load();

    // On closing the window, quit.
    v.onClose = function()
    {
        Application.quit();
    }
```

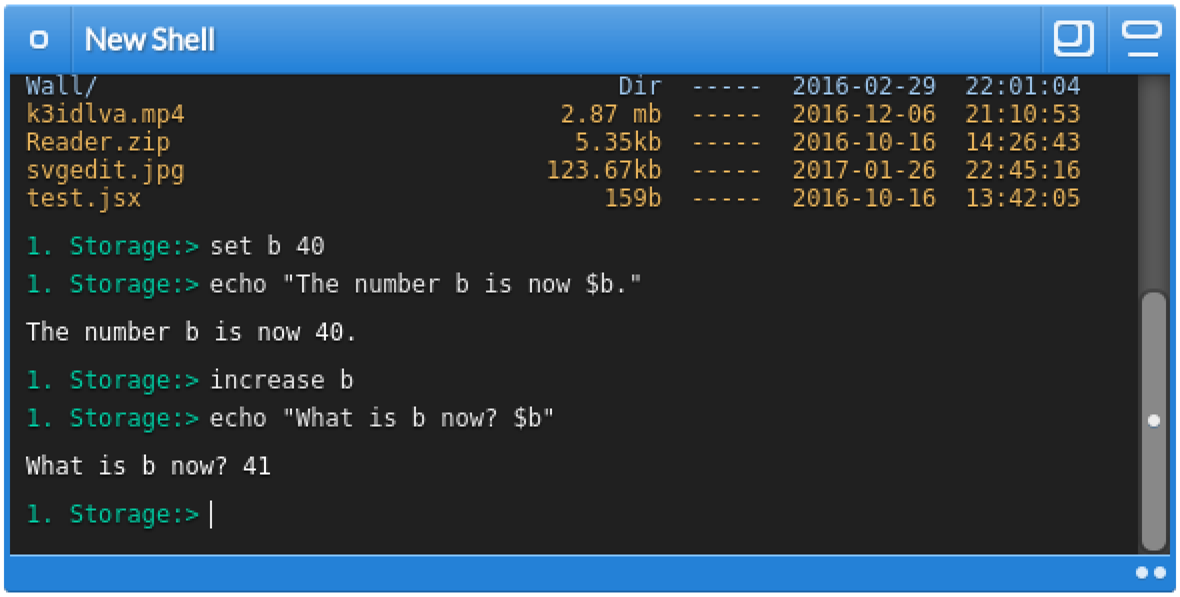
```
2. System:> cd Home:
2. Home:> cd Wallpaper/
2. Home:Wallpaper/> dir

wp_beach.jpg        wp_morerocks.jpg   wp_rocks.jpg
wp_coffee.jpg       wp_mountains.jpg
wp_microscope.jpg   wp_omnious.jpg

7 items total.


2. Home:Wallpaper/>
```

Friend Workspace core applications for development

The Friend Workspace comes with a short list of preinstalled applications that can be used for development. Some of the applications provide an API for other applications to interact with them. Some are simple tools. Here is a brief overview of the applications you may use in your development.

Friend Shell

Friend Shell is Friend's command line interface (CLI). It provides a comprehensive feature set for important system functionality. It allows a user to browse file systems, run applications and scripts, etc.



```
Wall/                                        Dir  -----  2016-02-29  22:01:04
k3idlva.mp4                              2.87 mb  -----  2016-12-06  21:10:53
Reader.zip                                5.35kb  -----  2016-10-16  14:26:43
svgedit.jpg                             123.67kb  -----  2017-01-26  22:45:16
test.jsx                                    159b  -----  2016-10-16  13:42:05

1. Storage:> set b 40
1. Storage:> echo "The number b is now $b."

The number b is now 40.

1. Storage:> increase b
1. Storage:> echo "What is b now? $b"

What is b now? 41

1. Storage:>
```

A CLI is a very useful interface for advanced users. It lets you issue precise instructions to your computer, and will allow you to automate several tasks. The CLI is the ultimate tool to get to the lower levels of the operating system. The naked GUI only gives you a "bird's eye view".

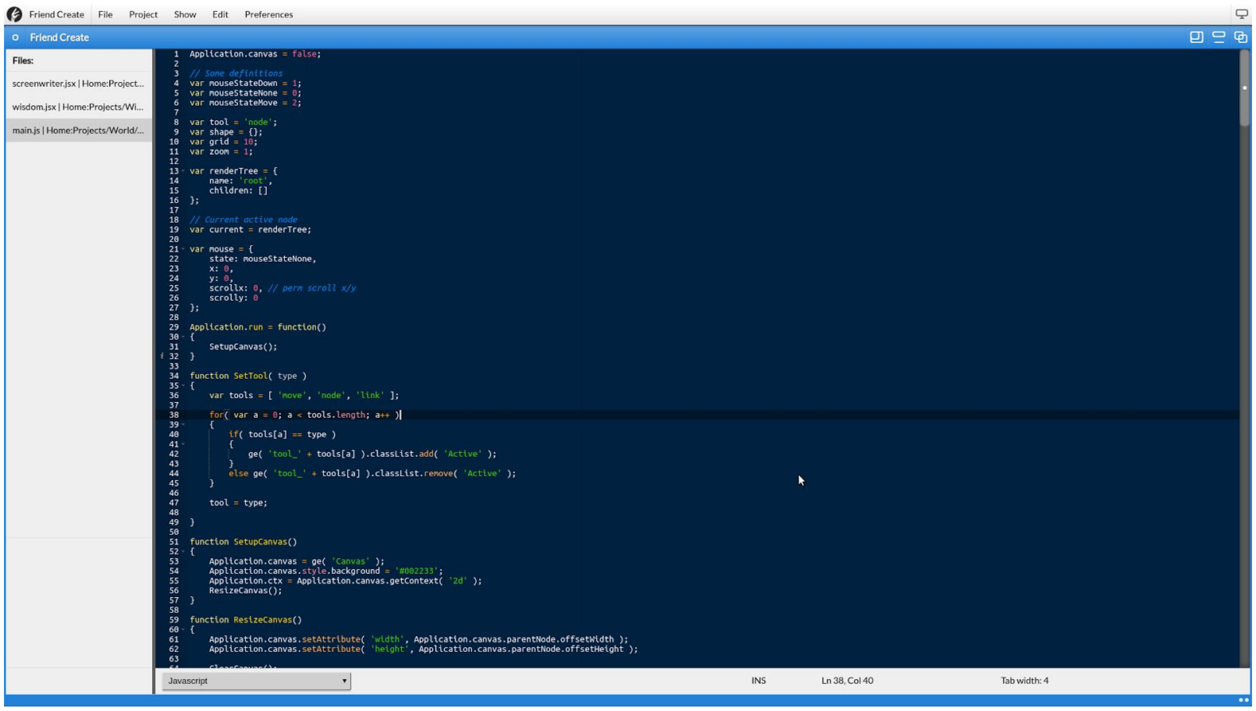
Advanced users are often found using a CLI to supplement their usage of a GUI. A CLI is expressive, while a GUI is implicit. A typical user usually interacts with the GUI through its offered default settings, templates, and constrained-selection gadgets. The advanced user can interact through the CLI with custom expressions from task to task, with complete freedom of how to use the available facilities.

For more on the CLI and using Friend DOS, please refer to the **Friend DOS** chapter.

Friend Create

Even though you may choose to utilize your existing programmers toolchain, we do offer a programmer's editor right inside of the Friend Workspace. It is called Friend Create, and is bundled with every FriendUP distribution.

Friend Create is a simple programmers editor, with only the main functionality you would expect in such a tool. But in addition to being able to handle code and projects, you may enjoy how it is integrated in the Friend system.

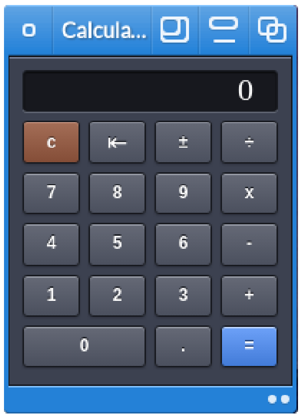


```
Application.canvas = false;

// Some definitions
var mouseStateDown = 1;
var mouseStateNone = 0;
var mouseStateMove = 2;

var tool = 'node';
var shape = {};
var grid = 10;
var zoom = 1;

var renderTree = {
    name: 'root',
    children: []
};

// Current active node
var current = renderTree;

var mouse = {
    state: mouseStateNone,
    x: 0,
    y: 0,
    scrollx: 0, // perm scroll x/y
    scrolly: 0
};

Application.run = function()
{
    SetupCanvas();
}

function SetTool( type )
{
    var tools = [ 'move', 'node', 'link' ];

    for( var a = 0; a < tools.length; a++ )
    {
        if( tools[a] == type )
        {
            ge( 'tool_' + tools[a] ).classList.add( 'Active' );
        }
        else ge( 'tool_' + tools[a] ).classList.remove( 'Active' );
    }

    tool = type;

}

function SetupCanvas()
{
    Application.canvas = ge( 'Canvas' );
    Application.canvas.style.background = '#002233';
    Application.ctx = Application.canvas.getContext( '2d' );
    ResizeCanvas();
}

function ResizeCanvas()
{
    Application.canvas.setAttribute( 'width', Application.canvas.parentNode.offsetWidth );
    Application.canvas.setAttribute( 'height', Application.canvas.parentNode.offsetHeight );

```

For more information about Friend Create, please refer to the User's manual.

Calculator

No desktop environment is complete without a calculator. In Friend, the Calculator application is a simple, yet handy app that allows you to calculate numbers.



The calculator is based off an open source MIT licensed version written in HTML5. It's one of the first applications that was "ported" to the Friend Workspace.

Friend Libraries

Libraries are runtime linked collections of executable code that are directly connected to the Friend Core. As such, they share memory space with Friend Core and have the best performance of any extendable Friend Core functionality.

In Friend Core, there are six main libraries:

- system.library
- mysql.library
- application.library
- image.library
- properties.library
- z.library

All of these libraries are available to a Friend developer (given the right access privileges). Some of the library functions are only available to administrators. But most are available to any user with a ***valid sessionid*** string.

system.library

The system.library in FriendUP is the extensible component that handles most of the logic in Friend Core. When it is extended, or even completely replaced, it can make Friend Core behave utterly different. Such a scenario might not be too far fetched, as you customize your Core for a different use. The system.library establishes the operating system template on the server core. It *upgrades* operating system features of the underlying OS (Linux or Windows) to behave like a Friend system.

Using devices

In Friend Core, devices are units connected to Friend Core using DOS drivers and DOS handlers. There are a few ways to manipulate these using the **device library calls** of the system.library.

mount – mount user device

Mounting devices authenticates and connects DOS driver based disk volumes to your Friend session. A mounted volume will show up in your mountlist. If it is a visible volume, it will show up on your Workspace.

Parameters:

devname – name of device which will be used in system, it must be
unique (**required**)
path – path to a local storage directory. This path will be the root to all files and directories in the device, if the DOS driver supports this variable (optional)
type – DOS driver type. Every time mount is called FC is trying
to find a suitable DOS driver by using the type parameter.
(optional)

Example call:

```
http://friendos.com/system.library/device/mount?sessionid=12345&\
        devname=Home&path=/home/user/&type=Local
```

Result:

```
stringified json object
```

On success:

```
{"response":"successfully mounted"}
```

On failure:

```
{"response":<error>}
```

unmount – unmount user device

Unmounting a device disconnects it from your Friend session. It becomes unavailable for reading and writing. You will still be able to see it in your mountlist in the unmounted section. If it had been displayed on the Workspace, unmounting will remove it from view.

Parameters:
devname – device name which we want to unmount (**required**)
Example call:

```
system.library/device/unmount?sessionid=e92&devname=TEST
```

Result:

```
stringified json object
```

On success:

```
{"response":"successfully unmounted"}
```

On failure:

```
{"response":<error>}
```

list – return list of mounted devices.

To get a list of all of your mounted devices, you can run the list command. It will return a JSON string with stringified objects.

Parameters:

Example call:

```
        system.library/device/list?sessionid=e92
```
Result:
```
        stringified json object
```
On success:
```
        {"Name","Home","Path":"Documents","FSys":"phpfs","Config":"","Visible":"1″ ,"Execute":"
"}, …..
```
On failure:
```
        {"response":<error>}
```

listsys – return list of available filesystems.

To get a list of all available file system types, you can run this command.

Parameters:
Example call:
```
        system.library/device/listsys?sessionid=e92
```
Result:
```
        stringified json object
```
On success:
```
        {"Filesystems":[{"Name","phpfs"},{"Name": …..},…..]}
```
On failure:
```
        {"response":<error>}
```

refresh – refresh FC device file structure (read settings from DB)

All devices in Friend Core are buffered to allow for fast transactions over the network between your client device and the server. Because of this, database changes aren't automatically detected by Friend Core. After a device has been altered, a refresh command should be issued to the Friend Core server to tell it to synchronize its buffers with the current state of the database.

Parameters:
devname – device name which will reread configuration from DB
Example call:
```
        system.library/device/refresh?sessionid=e92&devname=TEST
```
Result:
```
        stringified json object
```
On success:
```
        {"response":"device updated"}
```
Otherwise:
```
        {"response":<error>}
```

share – share device with other users across one server

All devices managed by Friend Core can be shared with other users. This allows users to collaborate on the same disk volumes.

Parameters:
– share – share device with another user.
– devname – device name which will be visible for provided user. In current version device must be mounted.
– username – name of user to who will have access to shared device
Example call:
```
        http://friendos.com/system.library/device/share?sessionid=12345&devname=Home&username=t
est_user
```
Result:
```
        stringified json object
```
On success:
```
        {"response":"device shared successfully"}
```
On failure:
```
        {"response":<error>}
```

Handling files

read – read a file from disk

One of the most often used file commands is the read command.

Parameters:
– path – the Friend path of the file
– mode - rb = read binary, r = read text, rs = read streamed
- offset - offset in the file to start reading from
- bytes - number of bytes to read
- download - indicates to download the file on the user's machine, value = 0 or 1

Example call:
```
        http://friendos.com/system.library/file/read?sessionid=12345&path=Home:myfile.txt&mode=
rs
```
Result:
```
        string
```
On success (read text mode):
```
        ok<!--separate-->Here is the text data.
```
On success (read binary mode):
```
        Here is the text data.
```
On failure (read text mode):
```
        fail
```

On failure (read binary mode):

```
	null
```

write - writes to a file on disk

The counterpart of read, writes data to the filesystem

Parameters
– path – the Friend path of the file
– mode - wb = write binary, w = write text
- data - the data to be written

Example call:

```
	http://friendos.com/system.library/file/write?sessionid=12345&path=Home:myfile.txt&mode
=w&data=123456789
```

Result:

```
	string
```

On success:

```
	ok<!--separate-->"FileDataStored" : "size_written".
```

On failure (cannot access the file) :

```
	fail<!--separate-->"Response": "No access to file"
```

On failure (missing mode parameter) :

```
	fail<!--separate-->"Response": "nmode parameter is missing".
```

copy - copy a file to another

This function copies a source file to a destination file.

Parameters
– path – the Friend path of the file to copy
– to - the Friend path of the destination

Example call:

```
	http://friendos.com/system.library/file/copy?sessionid=12345&path=Home:myfile.txt&to=Ho
me:Documents/myfile.txt
```

Result:

```
	string
```

On success:

```
	ok<!--separate-->"Response" : "0", "Written": "size_written"
```

On failure (cannot access the source file) :

```
	fail<!--separate-->"Response": "No access to source"
```

On failure (missing mode parameter) :

```
	fail<!--separate-->"Response": "No access to destination"
```

upload - uploads files to the Friend system

This function uploads a list of files from the user's browser to the Friend filesystem.

Parameters

Example call:

Result:

```
string
```

On success:

```
ok<!--separate-->"Uploaded files" : "number_of_files_uploaded"
```

expose - create a public permanent link to a file

If you just want to publicly share one single file in one of your Friend disks, you can use the expose command to handle this.

Once a file has been exposed, it creates a public link like this:

https://theroot.tree:2048/sharedfile/a322e944b1e4fffa0cd8cdb34da2ff72/test.jsx

Parameters
– path – the Friend path of the file

Example call:

```
http://friendos.com/system.library/file/expose?sessionid=12345&path=Home:myfile.txt
```

Result:

```
string
```

On success:

```
ok<!--separate-->{"hash":"a522ea44b1e4bffa0cd8cdb34da2ff72","name":"test.jsx" }
```

On failure (cannot access the source file) :

```
fail
```

conceal - make a publicly available file private

If you have made a file public previously, and you want to make it private, you can conceal it.

Parameters
– path – the Friend path of the file

Example call:

```
http://friendos.com/system.library/file/conceal?sessionid=12345&path=Home:myfile.txt
```

Result:

```
string
```

On success:

```
ok<!--separate-->
```

On failure (cannot access the source file) :

```
fail
```

Friend Modules

Modules are the counterpart to Libraries. They work on an identical system of messages and are independent parts of code that can be written in any available language supported by the platform (C, PHP, Python etc.). Modules are executed by Friend Core when called, and do not remain persistent in memory on the server.

Modules are more task oriented than Libraries. Libraries are used for general functionality like data access and manipulation. Modules are used extensively throughout applications and indeed the Friend Workspace. In your applications, you can split logic between server code and client code, where server code is handled by your modules, and client code is handled by Javascript in the Workspace. FriendUP makes an intensive use of the modules via internal messaging transmitted between the Workspace in the browser and the Friend Core in the cloud. All of the functions that get used by the system are available to the developer.

System.module

The system module contains all the major functions you need to program a Friend web application. This documentation will list the main calls, grouped by category.

Administration commands

setsetting

Stores data for a Web Application, allowing the retrieval of the application state between sessions.

Parameters

`setting` - the name of the settings to set
`data` - the data to set, can be a JSON string

Returns

ok or *fail*

This example is extracted from the calendar Web Application, it retrieves the settings of this application and sets calendar to its value.

```
var m = new Module( 'system' );
m.onExecuted = function( e, d )
{
	if( e == 'ok' )
```

```
		{
			RefreshSources();
			Application.mode = 'edit';
		}
	}
	m.execute( 'setsetting', {
		setting: 'calendarsources',
		data: Application.sources
	} );
```

getsetting

Returns the data set by the 'setsetting' system command.

Parameters
	`setting` - the name of the settings to recover
Returns
	the setting as a JSON string as it was set by setsettings

This example is extracted from the calendar Web Application, it retrieves the settings of this application and sets calendar to its value.

```
	// Get an existing one!
	var m = new Module( 'system' );
	m.onExecuted = function( e, d )
	{
		if( e == 'ok' )
		{
			var sources = JSON.parse( d );
			var str = '';
			sw = 2;
			// Refreshed
			Application.sources = sources.calendarsources;
			if( callback )
			{
				callback();
			}
		}
	}
	m.execute( 'getsetting', {
		setting: 'calendarsources'
	} );
```

proxyget

Uses Friend Core as a proxy to communicate with an external system over http or https.

Parameters

`url` - the url to connect to
… - more parameters following the url parameter are transmitted to the proxy and are
dependant on the destination

Returns

the data returned by the receiver, as an XML or JSON string

This example is extracted from the Treeroot code, where it communicates with a Treeroot server to extract some data:

```
var m = new Module( 'system' );
m.onExecuted = function( e, d )
{
        if( e == 'ok' )
        {
                var j = JSON.parse( d );
                if( j.response == 'ok' && j.data && j.data.length )
                {
                        console.log( 'Recovery data sent to: ' + j.data );
                }
                else
                {
                        console.log( j.code + ' : ' + j.reason + ' : ' + j.info );
                }
                Application.sendMessage( {
                        command: 'recover',
                        destinationViewId: msg.parentViewId,
                        data : j
                } );
        }
        else
        {
                console.log( 'Some error trying to recover account ... ', { e: e, d: d }
        );
        }
}
m.execute( 'proxyget', {
        url: 'https://store.openfriendup.net/components/register/recover/',
        Email: msg.data.username,
        Encoding: 'json’
} );
```

getlocale

Returns the locale entries for a Friend resource (an application or a driver).

Parameters

 type - DOS drivers is the only value supported in this version of Friend
 locale - default locale to revert if the current one is not supported

This example of code is taken from the DiskCatalog Web Application:

```
// Read our locale
Locale.getLocale( function( data )
{
	var m = new Module( 'system' );
	m.onExecuted = function( e, d )
	{
		if( e != 'ok' ) return;
		Locale.importTranslations( d );
	}
	m.execute( 'getlocale', {
		type: 'DOSDrivers', locale: data.locale
	} );
} );
```

languages

Gets a list of all the available locale languages in the system. There are no parameters.

Return value:

 A JSON list of all the available languages.

Example:

```
var m = new Module( 'system' );
m.onExecuted = function( e, d )
{
	if( e != 'ok' )
		return console.log( 'Severe error!' );
	var s = JSON.parse( d );
	if( !s.shortNames )
		return console.log( 'Severe error!' );
	var str = '';
	for( var a = 0; a < s.shortNames.length; a++ )
	{
		var cl = s.shortNames[a] == lang ? ' selected="selected"' : '';
		str += '<option value="' + s.shortNames[a] + '"' + cl + '>' + i18n(
		'i18n_locale_' + s.shortNames[a] ) + '</option>';
	}
	ge( 'languages' ).innerHTML = str;
}
m.execute( 'languages' );
```

listuserapplications

Gets all applications registered / activated for the user.

Returns

a JSON encoded string with the installed application path

```
function getApplications( callback )
{
	var m = new Module( 'system' );
	m.onExecuted = function( e, d )
	{
		if( e == 'ok' )
		{
			return callback( JSON.parse( d ) );
		}
		callback( false );
	}
	m.execute( 'listuserapplications' );
}
```

Calls available to users

tinyurl

Creates a new url that can be used to simplify complex urls in the Friend system. For example, public files have a long url with url variables. By using the tinyurl call, you can simplify this complex url into eight alphanumeric characters.

Parameters

`source` - url string

`expire` - boolean, 1 or *nothing*

Returns

A return code, “ok” or “fail”, and then a JSON explaining the response. If the response is positive, the JSON response is:

```
{"response":"url successfully created","hash":"3AB051DE"}
```

Example:

```
var m = new Module( "system" );
m.onExecuted = function( e, d )
{
	if( e == "ok" ) console.log( JSON.parse( d ) );
}
```

```
m.execute( "tinyurl", { source: "http://mysite.com/webclient/index.html" } );
```

Developing your own modules

You can create a module for your own needs in any language you want, as long as you implement the entry functions and the messaging system. We currently use modules written in C, PHP and Python.

Programming modules in PHP

To develop PHP modules, you need to have access to the Friend Core. If you do not have this access, please refer to the **Administrator's Guide** and set up your own Friend Core server. Friend Software Corporation offers development servers for Friend developers. Please go to https://friendup.cloud for more information.

If you are a PHP developer, you will be pleased to find that we offer PHP support using a small PHP runtime. Include this file in your code so that you can receive variables from Friend Core. A simple module looks like this:

```
<?php

// Get access to the logging object
global $Logger, $args;

// Include the friend runtime
require_once( "php/friend.php" );

// Add something in the log file located in build/log.txt
$Logger->log( "We are giving a response." );

// Give a response to Friend Core!
if( $args->command == "hello" )
{
	die( "ok<!--separate-->{\"response\":\"hello world\"}" );
}
die( "fail<!--separate-->{\"response\":\"no known command\"}" );

?>
```

Modules are called by Friend Core when an event is fired. This event could be triggered from the Friend Workspace, another module or another network event.

When developing modules, you store the module file in:

```
build/modules/mymodule/module.php
```

Modules are called from Javascript like this:

```
// Execute our "hello" command
var m = new Module( 'mymodule' );
m.onExecuted = function( e, d )
{
	if( e == "ok" ) console.log( d );
	else console.log( "Failed" );
}
m.execute( 'hello' );
```

The PHP database object

When writing PHP modules, we have provided you with a convenient database object that you can use to access your database. The database object can access both Friend Core's own SQL database, as well as any other SQL database that is available over the network.

Example PHP code:

```
<?php

// Just list out some cars from a database
$total = 0.0;
if( $rows = $SqlDatabase->FetchObjects( "SELECT * FROM `Cars`" ) )
{
	foreach( $rows as $row )
	{
		$total += $row->Price;
	}
}
echo $total . " is the price.";

?>
```

The PHP SQLDatabase class

Open(*$host*, *$user*, *$pass*)
Opens up a database connection to a host. All variables are strings. Host is either ip address and port, or host and port. Example:

```
<?php

$r = new SqlDatabase();
$r->Open( "myhost.domain.com:3306", "username", "password" ) or die( "trying..." );
$r->Close();

?>
```

Close()

Closes a database connection.

SelectDatabase(*$database*)
Selects a database on the database server to use. The database variable is a string.

Query(*$query*)
Executes an SQL query on the database. The syntax is MySQL.

FetchArray(*$query*)
Fetches a two dimensional array of rows using a query.

FetchRow(*$query*)
Fetches a single array of a row using a query.

FetchObjects(*$query*)
Fetches an array of objects using a query.

FetchObject(*$query*)
Fetches a single object using a query.

Flush()
Clears cache of the SqlDatabase and removes the last queries.

For more information about the database functions, please consider reviewing the source code of the SqlDatabase class.

The PHP Door class

The Door class is Friend's way to abstract disk volumes in PHP. The Door class can instantiate and abstract any mounted disk volume that is found on a Friend Core server. Each DOS driver that is written in PHP inherits from this base class. The Door class may be used where the File class is insufficient.

```
<?php
$door = new Door( "Home:" );
$door->createDirectory( "My Photos", "Home:Documents/" );
?>
```

Methods in the Door class

createDirectory(*$dirname*, *$path*)
Creates a directory under the specified path. The path must be a Friend path.

putFile(*$path*, *$file*)
Copies a file to a Friend path. The *$file* is file binary data. If unsuccessful, the method will return *false*. If successful, it will return *true*.

getFile(*$path*)
Returns a File object that has been loaded with a valid Friend path. If the path is invalid, the method will return false.

dir(*$path*)
Returns a directory listing in JSON format from a valid Friend path. If the path was invalid, the method returns false.

The PHP File class

The PHP File class abstracts files in Friend Core across DOS drivers. It is your unified interface to access files in Friend using PHP.

Methods in the File class

File(*$path*)
The constructor expects a valid Friend file path. Returns a File object.

GetContent()
Returns the content of a loaded File object. The data is expected to be in binary format.

SetContent(*$data*)
Sets content on a File object. The data is expected to be in binary format.

Load(*$path*)
Loads a file object by path.

Save(*$content*)
Saves a File to a Friend disk volume. The file needs a valid path. If the *$content* variable is passed, it will be used instead of the existing data available in the object. In other words, any data having been set with File::SetContent will be ignored. If the *$content* variable is not passed, the existing content that is buffered in the File object will be saved to disk.

Example:

```php
<?php
// Save a file
$f = new File( "Home:Testing.txt" );
$f->SetContent( "Hello world!" );
$f->Save();
```

```
// Load
$o = new File( "Home:Testing.txt" );
$o->Load();
echo $o->GetContent();
?>
```

Programming in the Friend Workspace

Most developers using the Friend Unifying Platform will focus their development efforts using Javascript and the Friend Workspace APIs. Friend offers developers the option of doing "backendless programming". This means that the needs or requirements of a typical developer are covered using the client side APIs available in the Workspace.

Friend Applications

Friend Applications are usually written in Javascript, the programming language supported natively by your web browser or browser technology. They use the ***Friend javascript classes*** and ***helper functions*** as a foundation and extend on these to build fully working applications.

As Friend follows an operating system template, each application is sensitive to things like localization, permissions and Friend file structures. This chapter will go through some of these things, and how they relate to a Friend application.

The Config.conf file

When starting out writing a Friend application that is prepared for system wide installation, you must create a config file. This file is called **`Config.conf`**, and is placed inside your application directory (i.e. Progdir:).

Here is an example file:

```
{
        "Name": "My Application",
        "API": "v1",
        "Version": "0.1",
        "Author": "Friend Software Labs",
        "Category": "Demonstration",
        "Init": "Scripts/my_application.js",
        "E-mail": "developer@friendup.cloud",
        "MimeTypes": { "mapl": "open %f" },
        "Description": "A sample application configuration file...",
        "Permissions": [
                "Door Local",
                "Module System",
                "Module Files"
        ]
}
```

This application indicates to Friend where its ***initial javascript file*** is located. In this example, the “my_application” javascript file is located in the “Scripts/” directory. This file is the one that is read first when executing the application (it is a *.js* file, not a *.jsx* - as Friend applications that are installed system wide do not use the *.jsx* suffix). Then it adds which ***permissions*** are required. The ***category*** determines in which software directory the application will be found in your system. The other attributes are fairly obvious. ***Mime-types*** are default file formats handled by the application, together with the command to open them when double clicking on such a file. ***API v1*** tells you that this application is using the first API available for interfacing with Friend. It’s also the API that this documentation is describing.

The Application object

Every application in Friend has an Application object. The Application object in a Friend application is the most important part of the application. It is generated automatically by the Friend Workspace environment when your application is first executed.

The Application object allows your program to communicate with the Workspace object as well as the application's sub components. Because of this, it is very important to learn about the various methods available in the object.

The Application object is generated when you are running a Friend application. This happens internally in the Friend Workspace. Because of this, you never have to declare this object. It's already there when you start out, and you will extend it and add your own methods and properties to it.

The most important method in the Application object is **run()**. This function is triggered when your application assets are finished loading and it is safe to start executing your application. It takes one argument, ***msg***, which gets arguments and other variables from the Workspace itself (for example command line arguments).

Friend Workspace works by using sandboxed application containers in the form of **iframes**. When you are running a Friend application, it starts by creating an initial iframe where your initial Friend Javascript is executed. Once you open up new screens and view windows, they also get **iframes**. Each of these view or screen iframes are initialized with standard Application objects and the Friend API. These Application objects can message each other using the postMessage Javascript function. This is how a Friend Workspace application works.

Each Friend application is decentralized into multiple Application objects. This is very powerful, and allows for applications that can work concurrently across multiple clients by transparently replacing postMessage with *websockets* or *http calls*.

A very simple Friend application example

This example shows how the run() method opens a *View* window and loads a template. You can find more information on both the *View* and the *File* objects in this document.

```
// This is the main run function for jsx files and FriendUP js apps
Application.run = function( msg )
{
	// Make a new window with some flags
	var v = new View( {
		title: 'Welcome to FriendUP!',
		width: 640,
		height: 500
	} );

	// Load a file from the same dir as the jsx file is located
	var f = new File( 'Progdir:Template.html' );
	f.onLoad = function( data )
	{
		// Set it as window content
		v.setContent( data );
	}
	f.load();

	// On closing the window, quit.
	v.onClose = function()
	{
		Application.quit();
	}
}
```

The example creates a new View with the title "Welcome to FriendUP!" and sets the views dimensions to 640x500 pixels. If the user's screen is smaller in any dimension, the view will adapt to the available space, e.g. on mobile phones.

After the view is created, a new File object is instantiated. The File object gets a template as parameter. The example refers to "**Progdir:**" which the Friend Workspace always maps to the directory the application is executed from. In this example, "**Template.html**" is also located in the same directory or folder as my_application.js, and that is the **Scripts/** directory. See the simple Template.html file below.

Next, a handler for ***onLoad*** is registered. The handler simply puts the received data as content of the View object. After the onLoad handler is registered the **load()** function is called to actually load the data.

The last step is to register an ***onClose*** handler on the View that quits the application once the View is closed.

File Template.html:

```
<div class="ContentFull Padding ScrollArea">
    <p>Hello world!</p>
    <p><button type="button" onclick="Application.quit()">Goodbye world!</button></p>
</div>
```

Communication between applications

Friend Workspace makes use of the **HTML security model**. It uses the ***postMessage*** feature of Javascript to allow communication between applications and between different Views of the same application.

The methods that the Friend Workspace API uses for this are `sendMessage` and `receiveMessage`. The Workspace controls which messages go where. Messages can be sent with or without a target application/View.

Example of **sendMessage**:

```
// Just send a message to the parent Application object.
function sendingAMessage()
{
        var o = { an: "object", to: "send" };
        Application.sendMessage( { command: 'hello', data: o } );
}
```

Example of **receiveMessage**:

```
// Just parse the received message
Application.receiveMessage = function( msg )
{
        // Don't treat noisy messages that do not adhere to our spec
        if( !msg.command ) return;
        // Ah we got our message!
        if( msg.command == 'hello' )
        {
                console.log( "We got a message: ", msg.data );
        }
}
```

Callbacks when messaging

When messaging between applications, using callbacks can be handy to trigger some code to run once a message has been parsed. An Example is passing a message to a view window and then triggering a callback. Keep in mind, the scope of the main application is the scope where you are executing your Friend application. The scope of the view window is a view that is opened and where a script has been loaded with its own Application object.

Here is the example:

```
// Scope of main Application object and add a callbackId that holds
// the callback that will quit the application once triggered
// (Progdir:example.jsx)
viewWindow.sendMessage( {
       command: 'callme',
       callbackId: addCallback( function(){ Application.quit(); } )
} );

// Scope of view window (Progdir:templates/view.html)
Application.receiveMessage = function( msg )
{
       if( msg.command == 'callme' )
       {
              // Send a message to the root Application object
              this.sendMessage( {
                     type: 'callback',
                     callback: msg.callbackId
              } );
       }
}
```

Methods in the Application object

The Application object has a few reserved methods that are useful for messaging and communication between the different system layers.

- `sendMessage(` *`messageObject`* `)` - sends a message to the Workdpace object. May be sent to a predetermined destination, like a specific GUI object, or to the Workspace object to be processed using system calls.

- `setApplicationName(` *`newName`* `)` - sets a new application name. This one is visible in the system task list, and will be the task name to manage, or *kill*.

- `setSingleInstance(` *`boolValue`* `)` - if the boolean value is set to true, a user will not be able to launch any additional instances of the application. If set to false, the application goes into its default state, allowing for multiple instances.

- `loadTranslations(` *`path, callback`* `)` - loads translations from a path, like “Progdir:Locale/”. Finds files like en.locale, fr.locale based on your current locale setting. After having loaded the translations, a callback may be run.

- `quit()` - terminates the application.

Callback functions in the Application object

When events occur, the Application object will execute a named callback function to handle the event.

- `receiveMessage( messageObject )` - when sendMessage has been issued, the message will be trapped by the receiveMessage callback function.
- `onQuit()` - when your application is killed or quitting, the onQuit callback function will be executed, allowing you to clean up before terminating the application.

Attributes in the Application object

- `applicationName` - the name of the application, visible in the system task list
- `authId` - the session ID for the particular application, used for communication with Friend Core
- `viewId` - the View window id containing the Application object

Localization

Each Friend application can be localized. You localize an application by populating its Locale/ directory with language files. The language that will be loaded is set system wide in the Language user preference application.

Example file names with English, Norwegian and Italian:

- `MyApplication/Locale/en.lang`
- `MyApplication/Locale/no.lang`
- `MyApplication/Locale/it.lang`

Each locale file is a colon separated list of keywords and replacements. Example:

```
# We can also comment our locale file
i18n_the_bunny                    : The bunny
i18n_quit                         : Quit
i18n_edit                         : Change it
i18n_window_title                 : My localized window
i18n_my_description               : Welcome to my localized application
```

The left hand side of the locale file has the keyword. This one is the same in each of the locale files in any language. On the right hand side, you write the actual language specific string that will show up in your Friend application.

To use the locale feature in Friend, you can use the locale features of the Friend javascript classes. In addition, there is a **i18n() function** that you can use in your applications to automatically translate a string.

```
// Open a window and show a translated string:
var v = new View( { title: i18n( "i18n_window_title" ), width: 320, height: 200 } );
v.setContent( i18n( "i18n_my_description" ) );
```

Translations from path

To load translations from a specific path when the application is running, use the Application.loadTranslations method, described in the Application section. Example:

```
// Load a translation and say something:
Application.loadTranslations(
        "Progdir:Locale/",
        function(){ Alert( i18n( "i18n_hello" ) ); }
);
```

Sharing application data between users

Friend allows you to create applications where users can work on the same data sets in real time. Friend uses ***websockets*** for optimal speed, so that e.g. multiplayer games or applications like instant messaging and whiteboards can be implemented easily.

The underlying technology layer to achieve this is called **Shared Application Session**.

Shared Application Session class

The SAS class is used to connect an application across users. It uses Friend's websocket to minimise lag and allow for applications to push data to other users.

The SAS class uses the **initiator of a session** as **pivot point for communication**. Other users' data is sent to the owner of the session and the owner can process and if applicable pass on that data to the other users.

SAS.invite(users, inviteMessage, callback)

The invite method is used to invite other users to a shared session.
The expected parameters are:

- `users` - array - Array of the usernames to invite ("user1", "user2", "user3", ...).
- `inviteMessage` - string - Message to display to users in invite dialog.

- `callback` - function reference - Reference to function that shall receive the result of the invite call(s) - will receive `false` as parameter if the user is not the session host.

SAS.remove(users, removeMessage, callback)

This method removes one or more users from a shared session.
The expected parameters are:

- `users` - array - Array of the usernames to remove.
- `removeMessage` - string - Message to display to users.
- `callback` - function reference - Reference to function that shall receive the result of the remove call(s) - will receive `false` as parameter if the user is not the session host.

SAS.getUsers(callback)

This method returns an array of the usernames that participate in a shared session.

The only parameter is the `callback` function that shall receive the data from the server.

SAS.send(event, username)

The send method sends an event to be shared with the other users. Username is an optional parameter available only to the session host; the event may be sent from the host only to the specified user. Other participant's events are always sent only to the session host, and thus username parameter is ignored.

- `event` - js-object - on the form:
  ```
  {
        type : 'event-name',
        data : <data>
  }
  ```
  - `type` - string - the name of the event the recipient is listening for
  - `data` - the data that will be passed on to the event handler

- `username` - string, optional - name of the single user that will receive this event.

SAS.on(event, handler)

The on method is used to register handlers for given events.
The expected parameters are

- `event` - string - the event that shall be handled
- `handler` - function reference - the function that shall receive these events

SAS.off(event)

Unregister a handler for the given event.
The only parameter is `event` - string - the event to unregister.

SAS.close()

Close a shared session. Can only be called by the host and will inform all participants about the session being ended.

Event overview

The SAS class has a couple of built-in events. In addition to that, each application can register its own set of events.

Events for host mode

- user-add
- user-remove
- user-list

Events for client mode:

- client-accept
- client-decline
- client-close

Application specific events

Application specific events can be registered for both the session host and participants. The same type of event can be used for both. Even though the host can send different event data to participants than he originally received himself for a certain kind of event. Look for the *draw* event in the example below.

Shared Application Session example app: Whiteboard

The Whiteboard app ~~**can be used to look at**~~ shows how the SAS class ~~**is used to allow**~~ enables an application to ~~**let**~~ allow several users to edit the same dataset collaboratively in real time. It is a simple drawing application that assigns each user a color and lets them draw on the same virtual whiteboard.

The application differentiates between two modes: host and client mode. The Application.run method in the class executes that check:

```
if( conf.hasOwnProperty( 'args' ) && conf.args.hasOwnProperty('sasid') )
{
	Application.sasid = conf.args.sasid;
}
else
{
	Application.isHost = true;
}
```

The SAS class will launch the application on invited users' Workspaces ~~with~~ who have the correct id parameter `Application. isHost` should default to false.

Depending on the mode, the application then registers for different types of events:

```
Application.bindHostEvents = function()
{
	Application.sas.on( 'client-accept', Application.clientAccepted );
	Application.sas.on( 'client-decline', Application.clientDeclined );
	Application.sas.on( 'client-close', Application.clientClosed );
	Application.sas.on( 'draw', Application.clientMessage );
}

Application.bindClientEvents = function()
{
	Application.sas.on( 'draw', Application.boardMessage );
	Application.sas.on( 'set-color', Application.setUserColor );
	Application.sas.on( 'user-add', Application.userAdded );
	Application.sas.on( 'user-list', Application.updateUserlist );
	Application.sas.on( 'user-remove', Application.userRemoved );
}
```

The two functions above register the relevant events for the two modes. The `draw` event is registered in both cases, but different handlers are chosen for the event. This allows the session host to verify and if necessary modify the data before passing it on to other participants.

Helper functions

Like most frameworks, Friend also provides a set of helper functions that are available to Javascript programmers. These consist of **Storage functions**, functions for **Encode/Decode of data**, and **DOM helper functions**. They help developers quickly measure and organize data objects and structures in their applications. These functions are available in any Friend application that utilizes API v1 or later.

Storage functions

SetCookie(*key*, *value*, *expiry*)
Sets a cookie value in the client browser. *key* and *value* are both strings. *expiry* is the amount of days before the cookie expires.
Has no return value.

GetCookie(*key*)
Retrieves the *value* of a cookie by *key*.
Returns the value if found. If not, returns *false*.

DelCookie(*key*)
Removes a cookie by *key*.
Has no return value.

Encoding and decoding of data

EntityEncode(*string*)
Encodes a *string* into HTML entities.
Returns an HTML encoded string.

EntityDecode(*string*)
Decodes a *string* from HTML entities.
Returns the character string decoded from the HTML entities.

DOM helper functions

SetCursorPosition(*element*, *position*)
Sets the cursor *position* in an interactive input *element* or contentEditable *element*.
Has no return value.

TextAreaToWYSIWYG(*element*)
Converts a textarea input element into a DIV element with a contentEditable attribute.
Returns *true* on success or *false* on failure.

Include(*scriptSrc*)
Adds a script element to the DOM and loads it. Only adds it if it has not already been added.
Returns *true* on success or *false* on failure.

ActivateScripts(*string*)
Extracts scripts from a *string* and adds them to the DOM, effectively running them.

Has no return value.

RunScripts(*string*)
Extracts scripts from a *string* and runs them.
Has no return value.

GetWindowWidth()
Returns the width of the browser window.

GetWindowHeight()
Returns the height of the browser window.

GetElementWidth(*element*)
Returns the width of a DOM element.

GetElementWidthTotal(*element*)
Returns the width of a DOM element, including margins, borders and padding.

GetElementHeight(*element*)
Returns the height of a DOM element.

GetElementLeft(*element*)
Returns the left position of an element in the browser window.

GetElementTop(*element*)
Returns the top position of an element in the browser window.

Localization functions

i18n(*string*)
Returns a translated string.

i18nAddPath(*path*)
Adds a path where Friend can find locale files. Returns nothing.

i18nReplace(*string*, *array*)
Searches through a string and replaces keywords found in an array with translations. Returns nothing.

i18nClearLocale()
Removes all translations from memory.

String manipulation functions

Trim(*string*, *direction*)
Strips away whitespace on either the left, the right or both sides of a string.

StrPad(*string*, *length*, *padder*)
Fills a string with a padder for a total length of the new string. Returns the padded string.

EntityEncode(*string*)
Returns a string encoded with HTML entities.

EntityDecode(*string*)
Returns a string where HTML entities have been decoded.

NumberExtract(*string*)
Takes a number found in a string and converts it to a float, double or integer. Returns the number.

NumberFormat(*string*, *decimals*)
Takes a string containing a number and formats the number with x amount of decimals. Returns the resulting string.

Dialog functions

Alert(*title*, *string*, *closetext*)
Pops up a View window with the title as window title and string as dialog message. An optional closetext string can be passed, overriding the default localized "Understood" button text.

Confirm(*title*, string, *callback*)
The confirm dialog pops up a View window with the title as window title, and string as dialog message. You then get two buttons, one that confirms, another that cancels. The result is sent back to the callback function.

NotifyMessage(*title*, *string, callback, clickcallback*)
Pops up a little bubble in the tray area of the Friend Workspace. This can be used to signal the user that something noteworthy has happened. This message will also be logged, for later review by the user. *callback* is a callback function that is run when the message has been displayed. *clickcallback* is a callback function that is run when the bubble is clicked by the user. This can be used to bring up a View window or Widget with more information or user interaction opportunities.

The View class

The View class is used to create windows in the Friend Workspace. The Workspace also has a Screen class to open new Screens for applications where this makes sense. In most cases, the View class is used to provider a user interface for an application.

```
// Make a new window with some flags
var v = new View( {
       title: 'Welcome to FriendUP!',
       width: 640,
       height: 500
} );
```

The code snippet above shows a couple of lines from the example application. A view is instantiated with a configuration object. The following properties are supported:

- `title` - String - no default - title of the View
- `width` - Integer - no default - initial width of the view
- `height` - Integer - no default - initial height of the view
- `mobileMaximised` - Boolean - default: false - maximize view in mobile view
- `maximized` - Boolean - default: false - maximize view to available screen real estate.
- `hidden` - Boolean - default: false - hide the view
- `invisible` - Boolean - default: false - make the view invisible
- `borderless` - Boolean - default: false - display the windows without border
- `resize` - Boolean - default: true - makes the windows resizable/fixed size - even a fixed size window will never be bigger than the available screen real estate
- `screen` - display this view window on the screen specified (object)

There are more options that the system uses internally. For application development, these are not relevant.

Methods in the View class

The View class provides the following functions to let developers set its contents:

- `setContent( content )` - sets content as the content of the View; content should be an HTML string. The content will be added as child to the body tag in the iFrame the View uses. Standard theme dependent Friend CSS is applied. The API is available to script references in the content. The `content` string will be stripped from inline script and style tags.

- `setRichContent( content )` - sets rich content in an additional iFrame in the View. Script tags are removed from the provided content string.

- `setRichContentUrl( url, base, appId, filePath, callback )` - sets the content to be an iFrame with the source defined by the `url` parameter. The API is not available. No Friend theme CSS is applied to the content.

- `loadTemplate( url )` - load a template from an URL. The API is not available to script references in the content**.**

- `setContentById( id, data, callback )` - set the content of a given node - node is identified by its id. The callback is executed once the content has been set.

- `getContentById( identifier, flag, callback )` - get the content of a node inside a View. The callback is executed once the content has been acquired.

- `preventClose( trueOrFalse )` - set value to prevent the app to close its view window. You can still kill the application to close the view window, but the close() function will no longer be able to affect the view window.

- `sendMessage( dataObject )` - sends a message to the main Application object that is located in the scope of the View window. Each View has its own Application object when it is using the API. The main application should keep track of its view so that it can send messages between them.

- `setMenuItems( jsonObject )` - sets the menu for the view window. The parameter it takes is in the form of a JSON object. Please read more in "Pulldown menus".

The View class provides a couple of interfaces to let an application react to user interaction on the view:

- `onClose` - fired when the view is closed. Either by click on the ***close button*** or by a ***function call*** from the Application that controls the view.

A View has a resize element that allows users to resize the view (if resize is not set to false). A View also has a title bar that allows the View to be dragged around on the workspace.

On the top of a View, a couple of buttons are available. The availability of the different buttons depends both on the theme used and on the end users device (desktop/mobile):

Close button - always available.
Minimize button - minimizes the view

Swap depth button - brings a view to the front or sends it to the back of the display stack. The workspace user may also single click on the View title bar to raise that View to the front (or 'top' of the display stack).

The Widget class

The Widget class is a bit like the View class, but has both a different visual appearance and a slightly different behaviour. Of course, a View window is meant to manage different GUI layouts in your application, while widgets are more generic. With widgets, you have complete visual control. A Widget may be half transparent, like the Dock, or completely opaque, like a View window. A current limitation is that it can only be opened on the main Workspace screen.

When creating a new Widget, this is the syntax:

```
var w = new Widget( {
        width: 400,
        height: 400,
        valign: 'bottom',
        halign: 'right',
        above: true
} );
```

This would create a Widget that is 400x400 pixels wide and high, aligned bottom right of the screen, always staying above View windows and other elements (except the screen title bar).

Here is a list of supported flags:

- `animate` - whether or not size and position changes should be animated
- `transparent` - set if the Widget background is transparent
- `background` - set if there is a background image or color
- `border-radius` - set if you want rounded corners, in pixels
- `width` - the width in pixels
- `height` - the height in pixels
- `top` - the y coordinate of the Widget, in pixels
- `left` - the x coordinate of the Widget, in pixels
- `valign` - vertical alignment, top, bottom or middle
- `halign` - horizontal alignment, left, center or right
- `scrolling` - if there should be scrollbars on overflowing content
- `above` - if the Widget should lay above other views and widgets
- `below` - if the Widget should lay below other views and widgets

Methods in the Widget class

The Widget class provides the following methods:

- `getWidgetId()` - gets the id for the Widget, used to pass with messages
- `getFlag( string )` - gets the flag value of the Widget, by name
- `setFlag( string, value )` - sets the flag value of the Widget, by name and value
- `setContent( string, callback )` - sets the HTML content of the Widget. You can add an optional callback function that will be executed once the content has been fully set.
- `raise()` - gives the Widget a higher z-index
- `lower()` - gives the Widget a lower z-index
- `autosize()` - makes the Widget change size to fit its content
- `close()` - closes the Widget and frees up memory

When an application quits, all of its widgets are automatically closed.

The Screen class

The Screen class is used to create new screens in the Friend Workspace. The screen class is there to help you organize your View windows on a separate spatial area, as an alternative to just opening a View window on the default Friend Workspace screen.

```
// Make a new window with some flags
var v = new Screen( { title: 'Welcome to FriendUP!' } );
```

The code snippet above shows a couple of lines from the example application. A screen is instantiated with a ***configuration object***. The following properties are supported:

- `title` - String - no default - title of the View
- `background` - Custom background image or color

There are more options that the system uses internally. For application development, these are not relevant.

Methods in the Screen class

The Screen class provides the following functions to let developers set the Screen contents:

- `setContent( content, callback )` - sets content as the content of the Screen; content should be an HTML string. The content will be added as child to the body tag in the iFrame the Screen uses. Standard theme dependent Friend CSS is applied. The API is available. The `content` string will be stripped from inline script and style tags. The callback is executed once the content has been set.

- `setRichContentUrl( url )` - sets the content to be an iFrame with the source defined by the `url` parameter. The API is not available. No Friend theme CSS is applied to the content.

- `loadTemplate( url )` - load a template from an URL. API is not available.

- `screenToFront()` - Makes this screen the front most screen.

- `sendMessage( dataObject )` - sends a message to the main Application object that is located in the scope of the screen. Each screen has its own Application object when it is using the API. The main Application should keep track of its screens so that it can send messages between them.

- `setMenuItems( jsonObject )` - sets the menu for the screen. The parameter it takes is in the form of a JSON object. Please read more in "**Pulldown menus**".

The Screen class provides a couple of interfaces to let an application react to user interaction on the screen:

- `onClose` - fires when the screen is closed. Either by click on the close button or by a function call from the Application that controls the screen.

File dialogs

A file dialog is a special class. It shows up in a View window and gives the result of user interaction in a callback. It is not a class that instantiates a reusable object. It is used in a disposable way. An example of use:

```
// Open a load file dialog and return selected files
var description = {
       triggerFunction: function( items )
       {
```

```
                console.log( "These files and directories were selected:", items );
        },
        path: "Mountlist:",
        type: "load",
        title: "My file dialog",
        filename: "",
        mainView: Application.windowId
}
// Open the file dialog view window
var d = new Filedialog( description );
```

As you can see, the file dialog takes a ***description object*** in the constructor. The dialog immediately appears. The object that is returned is irrelevant, and can be disposed of.

These attributes are available for file dialogs:

- **triggerFunction** - a callback function that will receive the result of the file dialog. This might be *false*, or an array of fileinfo objects.
- **path** - a Friend path. Mountlist: can be given if you just want to show the list of available disk volumes.
- **type** - *load* or *open* for loading files, *save* for saving a file, and *path* for just selecting a path
- **title** - the view window title for the file dialog
- **filename** - optional - if it is a save dialog, then you can preset what the proposed save filename should be
- **mainView** - optional - a file dialog can block a parent view window, so that you can not access it before a selection has been made. The value must be a valid windowId, which you will get from any view object or Application contained inside a view.

The File Class

The File class is used to access files in your Friend application. It is using the Dormant DOS kernel shell. Using the File class, you can load and save data using Friend paths. You can also post binary data to file names and call library functions on disk or volume based libraries.

Examples of use:

```
// Load a file from the Home: drive
var f = new File( "Home:template.html" );
f.onLoad = function( data )
{
        console.log( "This is the content: " + data );
}
f.load();
```

```
// Call a library method on a disk based library
var l = new File( "Home:Libraries/mylib.library" );
f.onCall = function()
{
       Alert( "All done!" );
}
f.call( "convertimage",
       {
              inpath: "Test:image.jpg",
              outpath: "Test:out.png",
              format: "png"
       }
);
```

Friend paths

Friend paths usually start with a given mount, like .e.g “Home:” or “OurWorkgroup:”. There are however a couple of prefixes with special meaning:

“**`Progdir:`**” - when running a JSX the “Progdir:” will always point to the directory the script is running in. That makes it easy to move complete application folders around without breaking their functionality.

“**`System:`**” the System: shortcut points to the web server root path. It is read only and allows one to include files from the filesystem inside Friend applications. This way e.g. icons from the gfx directory can be included:

```
System:gfx/icons/64x64/apps/accessories-text-editor.png
```

Methods in the File class

- `File( path )` - constructor. Takes a Friend path to initialize (optional when only using `save()`).
- `i18n()` - replaces all keywords found in the currently applicable locale file when it loads the content.
- `onError()` - should be overloaded. Is triggered when an error occurs when loading or saving data.
- `doReplacements()` - replaces registered keywords on loaded content.
- `load()` - loads data by the current path.
- `save( content, path )` - saves data (content) into a file at a Friend path. The path is optional. If it isn't set, it will use the path given in the constructor.
- `call( command, arguments )` - calls a library function on a library file. The command is a string keyword. The arguments are given in a key/value object structure.
- `post( content, filename )` - posts as an upload to a filename friend path with content.

- `addVar( key, value )` - adds variables that will be sent to the server when using `load()`, `save()` and `call()`.
- `onLoad( data )` - should be overloaded. Is executed with data as its first argument after `load()` has been called.
- `onSave()` - should be overloaded. Is executed after `save()` has been called.
- `onCall()` - should be overloaded. Is executed after `call()` has been called.
- `onPost()` - should be overloaded. Is executed after `post()` has been called.

Public variables in the File class

- `replacements` - holds a key value object of all keywords to be replaced when the file content is loaded.

The Door Class

As one of the "low level" classes in Friend, you use the Door class when you need extra precision when working with files. The Door class abstracts the DOS drivers directly, and operates on a disk volume.

Here's an example of getting the file information about a file using the Door class:

```
// Get a door object and get file information about image
var d = new Door( "Home:" );
d.dosAction( "file/info", { path: "Home:Myfile.jpeg" },
      function( data )
      {
            var res = data.split( "<!--separate→" );
            if( res[0] != "ok" )
                  return false;
            var d = JSON.parse( res[1] );
            console.log( "Filesize: " + d.Filesize );
      }
);
```

The Module Class

The Module class is used to abstract the Friend Core modules. A module is a structure in Friend Core that holds an amount of server functions. Each server function can take arguments and return some data to the user. Modules are powerful and can be written in any language, such as PHP, which is utilized in many of the core Friend modules. By allowing developers to extend Friend with scripted modules, they can rapidly implement features on the server.

Example of a module call:

```
// Test the help function call in the system module
var m = new Module( "system" );
m.onExecuted = function( returnCode, returnData )
{
	if( returnCode != "ok" )
	{
		Alert( "Could not get help." );
		return false;
	}
	Alert( "Help: " + returnData );
}
m.execute( "help" );
```

Methods in the Module class

- `Module( moduleName )` - constructor. Takes a module name as its argument. The module object will then initialize as an abstraction to that module, if it exists.
- `addVar( key, value )` - adds a variable to the module object. This variable will then be passed in the next module call.
- `execute( function, args )` - executes a module function call. The args variable is optional, and should be in the format of an object with key / value pairs.
- `onExecuted( returnCode, returnValue )` - should be overloaded. Is called once “execute” returns with a returnCode and/or returnValue.

The Library Class

The Library class is used to abstract Friend binary libraries. These run in server memory and give access to high speed functionality on the server. Said in a simpler manner, they allow you to ***use Linux or Windows binaries*** in your Friend Javascript application.

Example of a library call:

```
// Test a library call
var l = new Library( "system.library" );
l.onExecuted = function( returnCode, returnData )
{
	if( returnCode != "ok" )
	{
		Alert( "Could not call function." );
		return false;
	}
	Alert( "We got a directory listing: " + returnData );
}
l.execute( "file/getinfo", { path: "Home:" }  );
```

Methods in the Library class

- `Library( libraryName )` - constructor. Takes a library name as its argument. The library object will then initialize as an abstraction to that library, if it exists.
- `addVar( key, value )` - adds a variable to the library object. This variable will then be passed in the next library call.
- `execute( function, args )` - executes a library function call. The args variable is optional, and should be in the format of an object with key / value pairs.
- `onExecuted( returnCode, returnValue )` - should be overloaded. Is called once "execute" returns with a returnCode and/or returnValue.

FriendNetwork

FriendNetwork gives you the possibility to easily connect your Javascript application to any other on yours or other people's machine and exchange messages. A good example of the use of FriendNetwork is in the Shell, where the 'friendnetwork' commands allow you to be a host or access a distant host shell.
FriendNetwork can provide classic WebSocket connections between two machines, or faster peer-to-peer data exchange for games or large data transfers without having to pass through a server.

Principle

We have designed FriendNetwork so that it is as easy to use as possible from your Javascript application.

- You access FriendNetwork via FriendNetwork.method_name(parameters)
- You receive the result of this call as a message sent to the Application object that has done the call (in the receiveMessage method). The object transmitted in this message will have as properties :
  - command: 'friendnetwork'
  - subCommand: 'depending_on_call'

FriendNetwork API

FriendNetwork.host(hostName [, password])

Initiate a hosting session. Your host will be visible on the network.

- hostName: the name of the host to create, example 'MyGame', 'Charles Cave'. This name will appear when calling FriendNetwork.list.
- password: this optional parameter can be used when establishing non peer-to-peer connexions. If a client asks for permission to connect to your host, and if 'password' is not defined, he will have to return your own Friend password to be able to connect. If you provide a password, he will able to connect with both your own Friend password

or the provided one, thus enabling two types of connexions, one for administrators (you or anyone who knows your Friend password) and guests.

Once the host is established, FriendNetwork sends a message back to the application:

- command: 'friendnetwork'
- subCommand: 'host'
- key: the key to this host, that you should save for later use
- name: the full name of your host, in the form of 'host_name@user_name'

When a client connects to your host, you receives a message:

- command: 'friendnetwork'
- subCommand: 'cllientConnected'
- hostKey: the key of your host
- key: the key of the new session created for this client, different from the host key., You should save it. Ann unlimited number of client can connect to your host.
- name: the username of the the client that connected
- sessionPassword: true if the client provided your main Friend password (administrator), false if he used the guest password.

When a client disconnects himself from your host, you receives a message:

- command: 'friendnetwork'
- subCommand: 'clientDisconnected'
- hostKey: the key of your host
- key: the key of the client session
- name: the username of the person who disconnected

When a client sends a message to your host, you receives a message:

- command: 'friendnetwork'
- subCommand: 'messageFromClient'
- hostKey: the key of your host
- key: the key of the client session
- data: the data sent by the client

If an error occurred, an error message is sent:

- command: 'friendnetwork'
- subCommand: 'error'
- response: ERR_HOST_ALREADY_EXISTS or other connexion errors (see below)

FriendNetwork.setPassword(hostKey, password)

Changes or defines the guest password of your hosting session.

- hostKey: the key of the host
- password: a string containing the new password

If an error occurred, an error message is sent:

- command: 'friendnetwork'
- subCommand: 'error'
- error: 'ERR_HOST_NOT_FOUND'

FriendNetwork.dispose(hostKey)

Closes the host, making it invisible on the network and closing all communications. Every connected client receive a message allowing them to take action.

- hostKey: the key of the host session, as returned by the 'host' method

Once the host has been successfully closed, your application receives a message:

- command: 'friendnetwork'
- subCommand: 'dispose'
- hostKey: the key of the host closed
- name: the name of the host closed

If an error occurred, an error message is sent:

- command: 'friendnetwork'
- subCommand: 'error'
- error: 'ERR_HOST_NOT_FOUND' if the key is invalid or other connexion errors (see below)

FriendNetwork.connect(hostName)

Initiate a WebSocket connexion with a host.

- hostName: the name of the host to connect to. It can be in the form of 'host_name' (example 'Charles's Cave' or combined with the username of the hosting Friend machine (example 'Charles' Cave@charles')

If the connexion request has reached the host, your application receives a message:

- command: 'friendnetwork'
- subCommand: 'getCredentials'
- key: the key of this client session, to save for later use

You should answer to this request by calling FriendNetwork.sendCredentials with the proper password.

If an error occurs, an error message is sent.

FriendNetwork.sendCredentials(key, password)

Use this method to send the password to your host.

- key: the key of the client session
- password: a string containing the password

After calling this method, you can received two different messages.

If the password was incorrect:

- command: 'friendnetworkk'
- subCommand: 'wrongCredentials'

You have a limited time to send the password again until you will receive a 'credentialsTiimeout' message, signifying that the connexion is aborted.

If the password was correct:

- command: 'friendNetwork'
- subCommand: 'connected'
- key: the client key
- hostName: the name of the host
- sessionPassword: true if you have connected as an administrator with the host Friend password, false if you have connected as a guest with the secondary password

Once the 'connected"' message has been received you can start to send messages to the host, and you will receive his messages.

When you receive a message from your host, this message is transmitted to your application:

- command: 'friendnetwork'
- subCommand: 'messageFromHost'
- key: the key of the client session
- name: the name of the host
- data: the data the host sent to you

If the host has closed his connexion, you receive a message:

- command: 'friendnetwork'
- subCommand: 'hostDisconnected'
- key: the key of the client session that has been closed
- name: the name of the host

FriendNetwork.disconnect(key)

Closes a connexion with a host. The host will receive a message indicating that you have quit.

- key: the key of the client session, as returned by FriendNetwork.connect

If an error occurred, an error message is sent:

- command: 'friendnetwork'
- subCommand: 'error'
- error: 'ERR_CLIENT_NOT_FOUND' if the key is invalid or other connexion errors

FriendNetwork.send(key, data)

Sends a message to the distant application.

- key: the key to the communication session. This can be the key received in the 'connected' message if you are a client, or the key received in the 'clientConnected' message if you are a host. If you use the key of your host, as received in the 'host' message, the message will be sent to all the connected clients.
- data: the data to be sent. It can be a string or a Javascript object containing properties.

If the message is sent to a host, he will received a 'messageFromClient' message.
If the message is sent to a client, he will receive a 'messageFromHost' message.

FriendNetwork.closeApplication()

Call this method when you exit your application. It will close all hosts and clients and send the disconnexion messages to the distant sides.

FriendNetwork.status()

This method returns a list of currently open session of FriendNetwork for the current user.

After calling it, you will receive the following message:

- command: 'friendnetwork'
- subCommand: 'status'
- connected: true if connected to the server or false if not
- hosts: an array of the hosts currently present on the machine.
  - key: the FriendNetwork key of this host
  - name: its name (host_name@user_name)
  - applicationName: the name of the application that created the host (currently 'application')
  - applicationId: the identifier of the application that created the host
  - window: the window of the application that created the host
  - hosting: an array of the sessions currently hosted by this host, containing:
    - key: the FriendNetwork key of this session
    - distantName: the username of the client
    - distantAppName: the name of the application that connected
- clients: an array of the clients currently connected to distant hosts
  - key: the FriendNetwork key of the client
  - window: the window that established the connexion
  - hostName: the name of the host that it is connected to

- applicationId: the identifier of the application that established the connexion
- applicationName: the name of the application that established the connexion (currently 'application')

Errors

FriendNetwork errors are reported by a specific message to your application.

- command: 'friendnetwork'
- subCommand: 'error'
- error: string containing the name of the error
- key: the key of the session that sent the error, if applicable

'error' can have the following values

- 'ERR_HOST_NOT_FOUND': the host was not found in the list of hosts, or the hostKey provided was invalid
- 'ERR_CLIENT_NOT_FOUND': the client key was invalid
- 'ERR_TIMEOUT': connexion has timed-out, session 'key' is closed
- 'ERR_CREDENTIALS_TIMEOUT': the time allowed to send the correct password has exceeded the defined value. The connexion is refused and the session is closed.
- 'ERR_CONN_CLOSED' the connexion has been closed due to a network problem
- 'ERR_REQUEST_TIMEOUT': the network request got no reply from the network, indicating that the distant side may have abruptly disconnected

Programming GUIs

Friend implements a bare bones GUI toolkit based on HTML5 templates. At this moment in time, any rudimentary GUI layout is possible using the Friend GUI classes and helper functions.

Friend distinguishes between ***HTML5 templates*** and ***GUI logic***. When writing a GUI for Friend, one typically opens up a Friend screen or view window and then loads an HTML5 template into it. After that, you may want to run some Javascript that instantiates GUI objects on the template. In Friend, we offer quite a few GUI objects to play with.

How to write templates

When writing an HTML5 template for a Friend GUI, there are a lot of standard classes you can use to create a user friendly and appealing layout. These classes are ***theme-able***, and they are designed to be ***responsive*** for mobile devices. Additionally, they allow you to combine them with other ***CSS frameworks*** that you may prefer.

There are many different types of GUI layouts for different types of applications. Some applications need horizontal tabs. Some use vertical tabs. Some are more like dialogs or requesters. Depending on what you want to achieve, we will use these layout types to aid us in exploring the **CSS classes** needed for each one.

A simple layout with a bottom bar

Here is a simple layout where we allow a user to input a username and password. It uses simple CSS classes. Here we start out with the *ContentFull* class that encompasses the entire GUI. It has 100% width and height, starting from the top, left corner of the View window. Here, another class has been entered, *LayoutButtonbarBottom*, which assumes a layout where you have a content pane on the top, and a button bar pane on the bottom. Following this, we have the *VContentTop* class, which gives is the pane on the top. Then we have the *VContentBottom* class, which gives us the button bar on the bottom. The *VContentTop* also has the *ScrollArea* class, which makes sure there is a scroll bar if the content is higher than the area of the pane. The *Padding* class makes sure the default padded spaces are put in place in the content panes. The *BorderTop* class just puts a border to indicate where the button bar is positioned to the user.

For the content inside the *ScrollArea*, we have a *Padding* area with a simple strong heading. We could use H1-6 here, but in GUIs, we often just use bold or normal text elements. Inside we have an *HRow*, which stands for a *horizontal row*, meaning, we expect floating elements. These are defined with *HContentX* (where X is 5-100 for percent of the row width). The *FloatLeft* class indicates that the column field should float in the row container.

With *HRows* and *HContentX* elements defined, you can create a complex GUI inside a View window.

```
<div class="ContentFull LayoutButtonbarBottom">
	<div class="VContentTop ScrollArea">
		<div class="Padding">
			<p class="Layout">
				<strong>Enter username and password</strong>
			</p>
			<div class="HRow">
				<div class="HContent30 FloatLeft">
					<strong>Username:</strong>
				</div>
				<div class="HContent70 FloatLeft">
					<input type="text" class="FullWidth"/>
				</div>
			</div>
			<div class="HRow">
				<div class="HContent30 FloatLeft">
					<strong>Password:</strong>
				</div>
				<div class="HContent70 FloatLeft">
					<input type="password" class="FullWidth"/>
				</div>
```

```
                </div>
            </div>
        </div>
        <div class="VContentBottom Padding BackgroundDefault BorderTop">
            <button type="button">Save user!</button>
        </div>
    </div>
```

The horizontal tab layout

Here we are creating an application with a few tabs and a bottom bar with buttons.

```
    <div class="ContentFull LayoutButtonbarBottom">
        <div class="VContentTop ScrollArea">
            <div class="Padding">
                <div class="Pages" id="Mytabs">
                    <div class="Tab">Tab 1</div>
                    <div class="Tab">Tab 2</div>
                    <div class="Page">
                        <div class="Padding">
                            <p>This is the content of page 1.</p>
                        </div>
                    </div>
                    <div class="Page">
                        <div class="Padding">
                            <p>This is the page 2 content text..</p>
                        </div>
                    </div>
                </div>
            </div>
        </div>
        <div class="VContentBottom Padding BackgroundDefault BorderTop">
            <button type="button">Ok that was fun</button>
        </div>
    </div>
```

The content of the top pane is the tab layout. It is a series of classes that first define the tab area itself, then the pages that belong to the tabs. The tabs and the pages are associated in a chronological, ascending order. Tabs are described later in this document - they need to be initialized by Javascript in order to work.

A double column layout

Here, we are creating a GUI layout that has two columns, one of 60% width and another with 40% width. The left column has a negative background.

```
    <div class="ContentFull">
        <div class="HContentLeft HContent60 BackgroundNegative">
            <p><center>Left</center></p>
        </div>
        <div class="HContentRight HContent40">
            <p><center>Right</center></p>
        </div>
    </div>
```

Triple column layout using nesting

By using nested *HContentLeft/Right* elements, we can achieve three columns. Notice the *HContent66* and *HContent33* classes. They are specially made to obtain one or two thirds of a 100 percent for use in layouts. For the rest, you only have increments of 5 in the *HContent* class, like *HContent5* and *HContent55*. Each column is separated by borders.

```
    <div class="ContentFull">
        <div class="HContentLeft HContent66">
            <div class="HContentLeft HContent50">
                <p><center>Column 1</center></p>
            </div>
            <div class="HContentRight HContent50 BorderLeft">
                <p><center>Column 2</center></p>
            </div>
        </div>
        <div class="HContentRight HContent33 BorderLeft">
            <p><center>Column 3</center></p>
        </div>
    </div>
```

Vertical layouts

Just like we have HContentLeft and *HContentRight*, we also have *VContentTop* and *VContentBottom*, as shown earlier. These can be combined with *VContentX* to achieve vertical layouts - with nesting and all.

```
    <div class="ContentFull">
        <div class="VContentTop VContent80">
            <div class="HContentLeft HContent50">
                <p><center>Row 1, Column 1</center></p>
            </div>
            <div class="HContentRight HContent50 BorderLeft">
                <p><center>Row 1, Column 2</center></p>
            </div>
        </div>
        <div class="VContentBottom VContent20 BorderTop">
            <p><center>Row 2</center></p>
        </div>
    </div>
```

Lists

There are several list classes in Friend. But a typical list has a checkered background and rows with full width in respect to its container.

```
	<div class="ContentFull">
		<div class="ZebraList FullWidth">
			<div class="sw1">List item 1</div>
			<div class="sw2">List item 2</div>
			<div class="sw1">List item 3</div>
			<div class="sw2">List item 4</div>
		</div>
	</div>
```

This creates a list with full width, and checkered list items. If you want columns in these items, you need to add the *Columns* class, *HContentX* and *FloatLeft*:

```
	<div class="ContentFull">
		<div class="ZebraList FullWidth">
			<div class="sw1 Columns">
				<div class="HContent60 BorderRight Ellipsis FloatLeft">
					60% width list item 1a
				</div>
				<div class="HContent40 Ellipsis FloatLeft">
					40% width list item 1b
				</div>
			</div>
			<div class="sw2">List item 2</div>
			<div class="sw1">List item 3</div>
			<div class="sw2">List item 4</div>
		</div>
	</div>
```

Here, we are also using borders to separate the columns. In addition, we're using *Ellipsis* to make sure that overflowing text do not break the column layout.

A list of classes and their description

The classes in Friend are meant to be themable. Because of this, margin width or padding size may vary between themes. To make sure that the applications you make are theme compatible, you should stick to these css classes for the main layout of your GUI. You may of course add extra css classes where need be, but in that case, be mindful and remember to test your application in various themes before making it available to your users.

BorderLeft, *BorderTop,*	Default borders for each corner of an element separately.

BorderRight, BorderBottom	
BorderDefault	Combination of borders on each side of an element.
Rounded	Gives an element rounded corners.
PaddingTop, PaddingLeft, PaddingRight, PaddingBottom	Gives an element padding on a specific side of the element.
Padding	Gives an element padding.
MarginTop, MarginLeft, MarginRight, MarginBottom	Gives an element margin on a specific side of the element.
Margins	Gives an element margin on each side of the element.
Ellipsis	Intersect text overflow with three dots ("...")
ZebraList	Normal checkered list. Expects *sw1* and *sw2* classes on list items.
BackgroundLists	Negatively colored lists. Expects *sw1* and *sw2* classes on list items.

Mouse pointers

In Friend, there's a special mouse pointer setup to show various states of the Friend system, and the elements your mouse pointer is hovering over. Each element in a Friend GUI should have a css class that tells Friend in which state the element is. A state may be: ***busy, blocked***, ***clickable***, ***accurate***, ***movable***, ***typeable***, ***neutral***. Here is a list, with descriptions, of each mouse pointer state.

MouseDefault or none
This is the default mouse pointer state in Friend. In the default theme, the pointer is ***blue***, to indicate that you can click your mouse without activating any GUI element.

MousePointer

This class is set on an element to indicate to the user that it is clickable. By default, this turns the mouse pointer ***green*** when hovering over the GUI element.

MouseRestricted
This class is set on an element when it is not clickable, or blocked from interaction. By default, this turns the mouse pointer ***red*** when hovering over the GUI element.

MouseMove
When an element is movable, or draggable, this class turns the mouse pointer into ***arrows*** when hovering over that element.

MouseCrosshair
When an element, or area, has this class, the mouse pointer turns into a ***crosshair*** for aiming. This is perfect for graphic applications, where the user needs a precise tool to draw lines and other geometric shapes.

MouseCursor
Textareas, text input fields and content editable areas should be given this class to indicate that they are editable. By default, all *textarea* and *input[type=text|number]* fields are given this class.

Absolutely positioned elements

Sometimes, it is necessary to absolutely position elements to make them fully scalable. This must be done with more than an afterthought, as different themes may have different margins sizes and line heights etc. But if you need to absolutely position elements, you may use ***inline styling***. It is strongly urged to limit the use of such styling to the following keywords: ***top***, ***left***, ***bottom***, ***right***, ***width***, ***height***. Example:

```
        <div class="ContentFull">
                <div  id="Toolbar"  class="BorderRight"  style="width:  30px;  left:  0;  top:  0;
bottom: 0">
                </div>
                <div id="Canvas" style="left: 30px; right: 0; top: 0; bottom: 0">
                </div>
        </div>
```

You have been warned! Absolutely positioned GUIs often break when changing a theme, and you may end up getting support tickets that you could live without.

Pulldown menus

No application is complete without a functional menu. Well, even if you are against menus, they are an easy way to add access to functionality in a GUI application with the bare minimum of work.

In Friend, every View window or Screen has a method to add menu entries. Below is an example using a View window.

```
// Add a new view window
var v = new View( { title: "Test view", width: 400, height: 400 } );

// Create a menu
var myMenu = [
	{
		name: 'File',
		items: [
			{
				name: 'Quit',
				command: 'quit'
			}
		]
	},
	{
		name: 'Second menu',
		items: [
			{
				name: 'Say hello!',
				command: 'say_hello'
			}
		]
	}
];

// Add the menu to the view window
v.setMenuItems( myMenu );
```

So a Friend menu consists of a nested array of objects. Each object has a ***name***. Each object has either an ***items*** **array** or a ***command*** **string**. The items array will create sub menus. You can have several sub menus. The command string sends a message to the *Application.receiveMessage* function, where you can recognize it by testing the *msg.command* value in the message object.

A matter of scope

In addition to having a command string, there's the question of where the command is sent. By default, all menu commands are sent to the ***root Application object*** in a Friend application. Here it is intercepted by the *Application.receiveMessage(msg)* function. But this is quite often not what you would want.

A menu is set, either on a *View* window or on a *Screen*. Both have their own scopes, running code in an iframe nested in each one. The *root Application object* has got an invisible iframe running its code in a sandboxed environment.

To pass a menu command to the scope of its own iframe, you need the ***scope*** parameter set to ***local***. Like so:

```
{
	name: "Do some stuff",
	command: "do_stuff",
	scope: "local"
}
```

This will make sure that the command, "do_stuff", is sent to the scope of the *View* window or the *Screen* that it belongs to.

Tabs

In Friend, an HTML template adhering to the Javascript specification of tabs can be activated to an interactive tabbed interface using the following code:

HTML5:

```
<div id="MyTabs">
	<div class="Tab IconSmall fa-alert">Say hello</div>
	<div class="Tab IconSmall fa-minus">Less text</div>
	<div class="Page">
		<p>This is just a message to say hello.</p>
	</div>
	<div class="Page">
		<p>Told you it was less text.</p>
	</div>
</div>
```

It is important to note that *IconSmall* and *fa-** classes are added to the tabs. These are optional, but add icons in front of the tab labels. This can beautify the tabs, and doing it this way is strongly encouraged. To standardize, Friend utilizes Font **Awesome** for css compatible icons.

Javascript:

```
InitTabs( ge( 'MyTabs' ) );
```

The code above initializes the HTML5 template into becoming interactive and properly laid out. It is important to have loaded the HTML5 code into an element that is in the same scope that the Javascript is running in. Often times, you can embed the Javascript in your HTML5 template. Another way to accomplish the same thing is to load it using an external script reference in the HTML5 template:

```
<script src="Progdir:folder/myscript.js"></script>
```

Progdir, as explained earlier, is a relative path to your application directory on your file system.

Tree views

Tree views are useful when you want to hierarchically display lists that represent for example a data structure or a document. Tree view objects generate a DOM node that can be attached to an HTML5 template:

HTML5 template:

```
<p>Look at my fine tree view!</p>
<div id="WhereMyTreeViewIs">
</div>
```

Javascript:

```
var list = [ "Mercedes", "Volkswagen", "BMW", "Porché", "Saab", "Bugatti" ];
var tvi = Treeview( list, "brands", { alphabetical: true } );
tvi.id = "Car_brands";
if( ge( "WhereMyTreeViewIs" ) )
{
	ge( "WhereMyTreeViewIs" ).appendChild( tvi );
}
```

Directory views

Directory views are useful when you want to represent a directory or path on a file system in your graphical user interface. Directory views can be used on parent elements or directly on view windows or screens. Our first example below shows the use of a directory view on a View object.

```
// Create a new view window
var v = new View( {
	title: 'View test with directory view',
	width: 640,
	height: 480
} );

// Set up the directory view on the view window
var w = new DirectoryView( v );
```

Programming the Friend Core

The Friend Core is the Friend operating system kernel. It is designed as a cooperative system program that can extract functionality from an underlying operating system and expose it using Friend's APIs and structures. This allows you to work uniformly on top of any supported operating system.

When writing new components for Friend Core, it is advised to employ a scripting language like PHP. This allows you to write safe and solid code without having to worry about system crashes and memory bugs. Additionally, you get the added benefit of not being tied to changing system architectures on the underlying operating system.

Friend Core has a vast API that is accessible using HTTP and websockets. Because of this, you may want to use a different programming language that is more to your liking. But if you do choose to use PHP, there is a whole runtime environment available with classes and helper functions to get you started.

The first part of this documentation will explore the HTTP API of Friend Core. The second part will go into programming using PHP. Finally, there will be a chapter for C programmers who would like to examine Friend Core's C based libraries.

Getting started with Friend Core and HTTP/S

To get started, we will be utilizing CURL to access the Friend Core. This way, you will be able to learn incrementally using a simple tool that is available on any operating system. In the following examples, we are using Linux with CURL.

To log into Friend Core, you first need a username and password. It is quite easy to log into a standard Friend server using the following query:

```
	curl http://friendcore.local:6502/system.library/login/ -F "username=xxxxxxx" -F
"password=xxxxxxx" -F "deviceid=myid"
```

Remember to make sure your password is pre-hashed. If not, Friend Core won't be able to interpret your password. Passwords are hashed like this:

```
	PASSWORD = "HASHED" + sha256( "mypassword" )
```

The **device id** can be set to anything. It is a way for you, as a user, to recognize which unit is connected and logged in to your user account. A device id may f.ex. be: “MyPHPScript”.

It is important that you use curl in POST mode. Friend Core does not support the GET method for authenticating. If your username and password is correct, you will get a reply like this, in JSON:

```
{
        "result":"0",
        "sessionid":"9ba8866ba866da295a861913f9d2f7c6de1e28a1",
        "userid":"1",
        "fullname":"Hogne Titlestad",
        "loginid":"9ba8866ba866da295a861913f9d2f7c6de1e28a1"
}
```

You will now be able to use the rest of the API by adding your **session id** to other calls.

If you did not manage to login because of a bad username and/or password, you will get this message:

```
{
        "result":"-1",
        "response":"username and/or password not found!"
}
```

Friend Core has several libraries and modules that all are available using HTTP/S calls using Curl (or another web crawler). Please revert to the Modules and Libraries chapters for an overview of all the available commands that are available for these. But to show you how to use these, take a look below for some examples:

A library call using system.library:

```
curl http://friendcore.local:6502/system.library/help \
        -F "sessionid=9ba8866ba866da295a861913f9d2f7c6de1e28a1"
```

Remember to add options to the different library calls in adherence with the library documentation in the **Libraries** chapter.

A module call using the system.module:

```
curl http://friendcore.local:6502/system.library/module/ \
        -F "module=system" -F "args={\"setting\":\"settingtest\"}" \
        -F "command=setsetting" \
        -F "sessionid=9ba8866ba866da295a861913f9d2f7c6de1e28a1"
```

For modules, arguments are added in a JSON string with the args variable. So make sure you preprocess your arguments in the JSON format to be compatible with the Friend Core module format. Each module command is given as the command argument. Again, all queries to Friend Core need a **sessionid** string for authentication.

NB: If you are using HTTPS on **localhost** with curl, remember the --insecure command line option. It will tell curl to skip verifying your self signed certificate.

DOS Structures

The Friend DOS layer, Dormant, operates on file information structures in the JSON format. These structures are given when doing directory views or getting file information through "info" DOS calls. The JSON structures follow a specification with possible attributes.

Each file type is accessible in the **Type** attribute. There is also an extra **MetaType** attribute for determining what type of content the file has. For example, a meta type may be "News item". In that case, the file consists of many blocks of data that have been combined into one binary file. Such files can be read using the ***InfoGet*** and ***InfoSet*** methods (more on them elsewhere in the documentation, under DOS commands).

Each directory or file may have an **ID** attribute. This is optional, but may expose a primary key or identifiable key to a Friend application.

Path has the file system path *relative to the volume name*. Volume names are omitted from the paths.

DateCreated and **DateModified** are represented in the following format: **Y-m-d H:i:s**.

Filesize contains the size of the file in bytes. Directories have no file size (=0).

Permissions has the following format:

```
{
        User: "arwed",
        Group: "arwed",
        Others: "arwed"
}
```

Friend has a permission system inspired from Tripos and Amiga OS. Each letter is explained here:

```
A = Archive (the file is being used, stay away)
R = Readable
W = Writable
```

```
E = Executable
D = Deletable
```

Missing permissions are written as such, where write and delete are missing: “ar-e-”

Shared tells how the file is shared to others. The default is “Private”. If the file is shared, it may be “Public”. A **SharedLink** is accessible to users of the right privilege. Public files have public shared links that can be used by anyone.

Example of a file structure

A plain DOS file structure has the following format:

```
{
        DateCreated:"2017-04-06 09:30:33"
        DateModified:"2017-04-06 09:30:33"
        Filename:"3D-Hartwig-chess-set-master.zip"
        Filesize:"12038549"
        ID:"11914"
        MetaType:"File"
        Path:"3D-Hartwig-chess-set-master.zip"
        Permissions:"{‘User’:’arwed’,’Group’:’-----’,’Others’:’-----’}”,
        Shared:"Private"
        SharedLink:""
        Type:"File"
}
```

Example of a directory structure

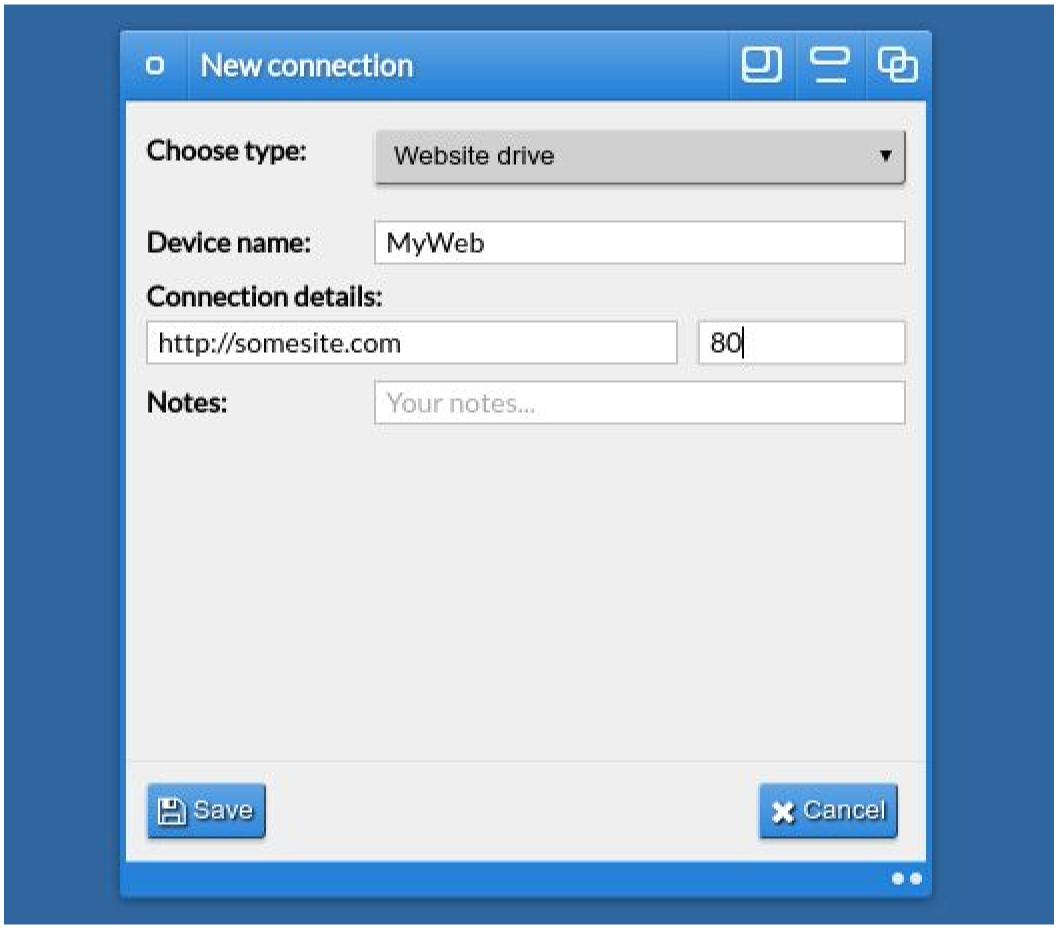
Directory structures are not unlike File structures. They have the following format:

```
{
        DateCreated:"2017-01-23 15:39:06"
        DateModified:"2017-01-23 15:39:06"
        Filename:"3D-Hartwig-chess-set-master"
        Filesize:"0"
        ID:"725"
        MetaType:"Directory"
        Path:"3D-Hartwig-chess-set-master/"
        Permissions:"{‘User’:’arwed’,’Group’:’-----’,’Others’:’-----’}”,
        Shared:""
        SharedLink:""
        Type:"Directory"
}
```

Using DOS drivers

The Website DOS driver

The Website DOS driver allows you to connect to an **external Web application** and abstract it as a disk volume. This way, you can distribute your application by allowing users to connect to it through the DOS driver.


New connection

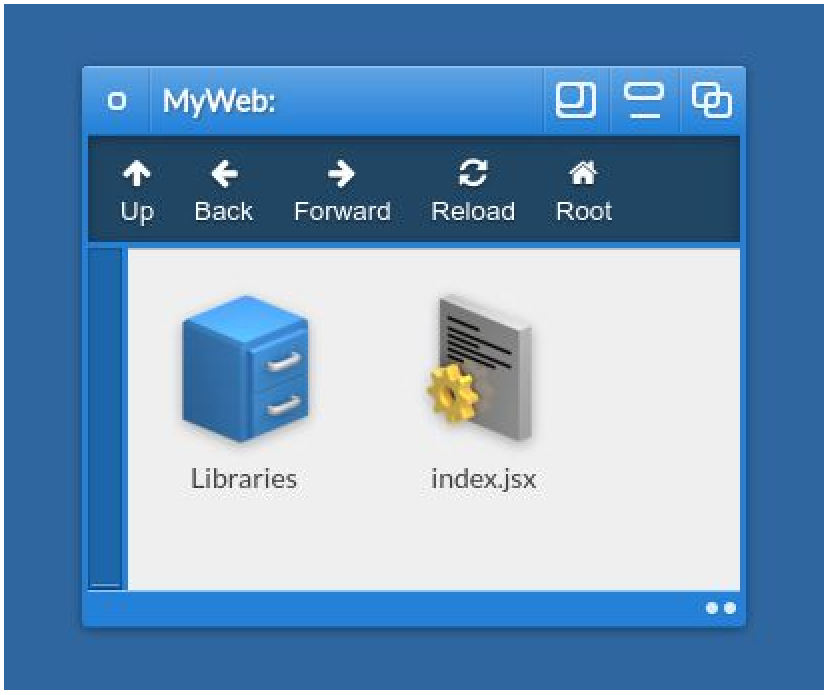
Choose type: Website drive

Device name: MyWeb

Connection details: http://somesite.com 80

Notes: Your notes...

The Website DOS driver supports not only directory listings, but also virtual disk libraries. This allows you to expose your application's server API as a Friend library. This way, your javascript application can call the API using library calls to access more functionality and data from the remote source.



When mounting a website drive, you will get a simple directory listing. You will get either an index.html file (if the website you are abstracting has such a file), or you will get an index.jsx file (if the website has made available such a file). The index.jsx file takes precedent, and if it's there, the index.html file will be hidden. The Libraries/ directory will list your virtual libraries.

For our example here, we are expecting you to use PHP for developing the external service for the DOS driver. In fact, when using the Website DOS driver, you do not really have to develop the logic of the DOS driver itself - it is already there. All you need to do is to handle the calls that come in from Friend Core on the remote server.

We have added these files to our remote server, hosted on Apache 2:

- index.jsx
- index.php
- templates/
    - main.html

These three files include everything we need for the moment. The **.jsx** file is the executable javascript that will be loaded by Friend Workspace. It will open a new window and display a template. The code is here:

```
Application.run = function( msg )
{
	var v = new View( {
```

```
                title: 'My sample Website app',
                width: 500,
                height: 400
        } );

        v.onClose = function()
        {
                Application.quit();
        }

        var f = new File( 'Progdir:Libraries/templates.library' );
        f.onCall = function( e, d )
        {
                if( e == 'ok' ) v.setContent( d );
        }
        f.call( 'template', { templateFile: 'main' } );
}
```

The **index.php** file includes the logic to receive the library call and to display the library. This is how the code looks:

```
<?php

// Check commands from Friend Core
if( isset( $_REQUEST['command'] ) )
{
        switch( $_REQUEST['command'] )
        {
                // Library listing?
                case 'libraries':
                        $o = new stdClass();
                        $o->Filename = 'templates.library';
                        $o->Filesize = '16b';
                        $o->DateModified = date( 'Y-m-d H:i:s' );
                        $o->DateCreated = $o->DateModified;
                        $o->IconClass = 'TypeLibrary';
                        $o->Permissions = '-r-e-';
                        // Return one file, a json encoded array of a file object
                        die( 'ok<!--separate-->' . json_encode( array( $o ) ) );
                        break;
                // A library call?
                case 'call':
                        if( !isset( $_REQUEST[ 'path' ] ) )
                                die( 'fail<!--separate-->{"response":"no library
specified"}' );
                        $library = end( explode( '/', $_REQUEST[ 'path' ] ) );
                        // This library?
                        if( $library == 'templates.library' )
                        {
                                if( !isset( $_REQUEST['args']['query'] ) )
```

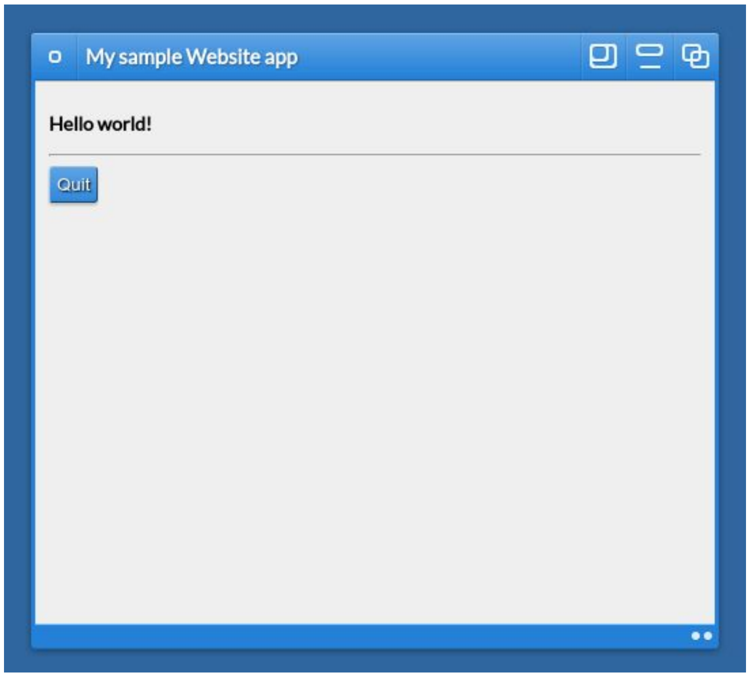
```
                                    die( 'fail<!--separate-->{"response":"no library
        call specified"}' );
                                switch( $_REQUEST['args']['query'] )
                                {
                                        // We want a template
                                        case 'template':
                                                // No traversal
                                                $tf = isset( $_REQUEST[ 'args'][
        'templateFile' ] ) ? $_REQUEST[ 'args'][ 'templateFile' ] : '';
                                                if( !$tf || strstr( $tf, '..' ) )
                                                        die( 'fail' );
                                                if( file_exists( 'templates/' . $tf .
        '.html' ) )

                                                {
                                                        die( 'ok<!--separate-->' .
        file_get_contents( 'templates/' . $tf . '.html' ) );
                                                }
                                                break;
                                }
                        }
                        break;
                // Unknown call
                default:
                        die( 'fail' );
        }
}

// Nothing was triggered...
die( 'fail' );

?>
```

The **templates/** directory holds our main.html template file, which is a simple template file for our GUI. It looks like this:

```
<div class="ContentFull Padding">
        <p><strong>Hello world!</strong></p>
        <hr/>
        <button type="button" onclick="Application.quit()">
                Quit
        </button>
</div>
```

NB: as you can see in the template, the Quit button has an onclick action that triggers Application.quit(). This function is built into every Friend GUI template, together with a bare minimum of functions. Read more in the section for the **Application object**.

When the index.jsx file is run, this is the resulting application:



You can keep adding calls to your library, and you can consider adding more libraries still. A Friend application hosted on a Website DOS driver can use all the same functionalities, functions, classes and Friend features as any other Friend application. This is because the templates in the application get access to the Friend Workspace API automatically.

The Server DOS driver

If you are an administrator, you can use the Server DOS driver to access a directory on your Linux or Windows server. By entering in a server path, you will get access to all the files and subdirectories contained within once you mount the volume in Friend.

The Server DOS driver is ideal when you want to use your existing development environment through SFTP to your Linux server. Every time you save a file, it will be updated inside the Friend Workspace, allowing you to access it without having to worry about cache.

Terminology (alphabetized)

The Friend Unifying Platform has a plethora of terms and expressions throughout this documentation, and indeed throughout the OS and GUI's themselves. To give a bit of insight, we have compiled a list of important terms and / or frequently occurring words, with an explanation below.

Alert() -

Amiga OS - The operating system created by Commodore for the Amiga computers. Was based on Motorola 68k processors and custom chips. One of the first pre-emptive multitasking operating systems.

Amiga DOS - The component of Amiga OS that was derived from Tripos.

API - Application Programming Interface

Application - See also **Modules** and **Libraries**.

Application category -

Application message, or msg -

Application object, *root Application object* - The Application object is generated when you are running a Friend application. See also the **Workspace object, View object,** and **Screen object)**.

Application Specific Event -

AREXX, REXX - Programming language for controlling applications. In Friend, our equivalent is called Dormant.

ASCII, UTF-8 - Encoded text data.

Authentication -

Callback, callback function - See also **Messaging** and **Event handler**.

Certificate - A document used for authentication. For example, an SSL certificate.

Class -

Client mode - Session client can register client-accept, decline, and close events. See also **SAS class**, **Host mode** and **Events**.

CLI - Command Line Interface - An interface that allows you to use the DOS.

CLI argument(s) - Strings or parameters typed following the shell **command**. These arguments are passed to the Friend Core to elaborate on, qualify, or modify the command execution.

Configuration file, Config.conf - see **Project configuration file**

console.log() -

Constructor -

Cookie -

CSS frameworks, classes -

CURL -

Database object - See **PHP database object.**

Device ID -

Disk Drive - shortened to 'disk' or 'drive'. Early computer storage devices, as well as current HDDs (Hard disk drives) consist of a magnetized circular metal surface (disk) that is rotated at high speeds by an internal (drive) motor. SSD's or Solid State (disks or drives), by contrast, have no internal motor or moving parts. Solid-state refers to integrated circuit (silicon only) ICs or computer memory chips. Today, even when persistent computer 'storage' is offered over the cloud, as a 'virtual' storage device, it is still referred to with the legacy name of 'Disk' or 'Drive'. When you hear that, just think 'storage'. And of course, there are many flavors...

DOM - Document Object Model

DOM element -

DOS - Disk Operating System - An OS that is managed using commands and expressions.

Dormant - Our interface to leverage Friend DOS to and between Friend applications and Friend Core, the Friend server/kernel.

Encryption - data may be encrypted - or - scrambled according to algorithmic rules so that it may remain privy to people who possess the encryption key only.

Events -

Event handler - A function that handles the processing of or response to system, application, or user events.

File class -

File dialog -

File object, fileinfo object -

Filesystem - A data volume that is structured so that it can be read by an operating system.

Filesystem driver - A driver that reads a data volume and interprets its file system for an operating system.

Friend binary libraries -

Friend Core - The Friend server/kernel. This is the brain of Friend.

Friend Shell - This is a Command Line Interface that allows a user to access the Friend Core using commands and arguments.

Friend Script - Friend DOS commands and arguments sequenced in a file.

Friend DOS - Friend Disk Operating System.

Friend Create - Friend's integrated development environment, or IDE.

Friend Chat - Friend's instant messaging and video conferencing application.

Friend Workspace - Friend's dynamic desktop environment.

Friend server - another name for Friend Core.

GUI - Graphical User Interface

GUI objects -

Handler - see **Event handler**.

Helper functions -

Home: volume - A common name for the disk volume designated for a user's main files.

Host mode - Session host can register user-add, remove, and list events. See also **SAS class**, **Client mode** and **Events**.

HTML security model -

HTML5 templates -

I18n - i18n() string language translation function. See also **Localization**.

iFrame -

JS object, JSON object - See also **JSON.**

JSON - JavaScript Object Notation, a standard string format to represent data objects. See also **JS object.**

Kernel - The core of an operating system - where resources, users and other important data is managed and processed.

Library - A collection of function calls contained in or abstracted through one file. See also **Modules** and **Applications**.

Library class -

Localhost -

Localization, Locale/ - An Application's Locale/ directory holds language support files that define system and application keywords and message strings, and their language-specific (Norwegian, English, German, etc.) string replacements. See also **i18n() conversion function**.

Message, or msg -

Messaging - See also **Callback.**

Methods - See also objects and properties.

Mime Types -

Mountpoint - The unit that represents a disk volume.

Modes - See **Host mode** and **Client mode**.

Modules - Collections of functions that are running outside of the Friend Core memory space. See also **Libraries** and **Applications**.

Module Class -

MySQL / database - A program that holds vast amounts of data and from where data can be retrieved and stored with great speed and efficiency.

Network Event -

Node, node ID -

Objects - See also properties and methods.

OS - Operating system

Parameters (command or function) -

File and directory **permissions, privileges** - Specified for Owner, Group, and Others. Rules, controls, or protections for user and group access to specific files and directories. Access types include Read, Write, Execute, and Delete.

PHP -

PHP database object -

Process, Task - Programs that run in a space managed by the operating system

Progdir:, or **Program directory -**

Project -

Project configuration file - config.conf

Properties - See also objects and methods.

Python -

Responsive UI design -

returnCode, returnValue, returnData -

SAML - Security Assertion Markup Language

Sandbox, sandboxed application containers -

Scope (of namespace?) - Application, Screen or View scope.

Screen class - See also **View class.**

Screen object - See also **View object.**

Screen window, or Screen - See also **View**.

Session -

Session ID -

Session host, participants - See also **SAS, SAS class**. **Host mode**, **Client mode**.

Shared Application Session (SAS) -

SAS class -

Shell, shell process - A textual user interface to a functional system structure, like a desktop or a CLI.

System: volume - The virtual disk volume that represents the Friend operating system in the Friend Workspace.

Template -

Tripos - An ancestor to the Friend Unifying Platform.

UI - User Interface. One example is GUI (see above). Another is CLI (command Line Interface - or typing on a keyboard). A third is Voice UI, such as speech-to-text-to-speech, voice command input, and audio response or output. A fourth, still in early-early alpha, is direct brain-to-computer.

View class - See also **Screen class.**

View object - See also **Screen object.**

View window, or View - See also **Screen**.

Volume - A disk unit connected to a disk drive. A Volume contains directories and files. This is represented as the ***volume-name***, followed by a *colon* **':'**, for example, **Work:**

Websockets -

Workspace - see Friend Workspace above.

Workspace object - The Workspace object is the main structure representing the Workspace scope in Javascript. See also the **Application object**.

Workgroup - A group that users can be a member of that gains them access to shared resources. Often times, users get access to networked disk volumes by being a member of such a workgroup.

Xcellent - FriendUP is an Excellent programming environment!