Topthemen dieser Ausgabe

ISSN 1867-7991

Editorial

Social Networking

Schon seit längerer Zeit ist freiesMagazin in sozialen Netzwerken vertreten, hat dies aber nie groß bekannt gegeben. Grund dafür ist vor allem, dass die Konten nicht offiziell von der freiesMagazin-Redaktion eingerichtet wurden, sondern von Fans des Magazins.

Dennoch haben wir uns entschieden, auf unserer Webseite auf der rechten Seite diese Konten aufzulisten, sodass man freiesMagazin leichter finden kann. Neben prominenten Vertretern wie Facebook [1] und Google+ [2] ist freiesMagazin auch über Nachrichtendienste wie Twitter [3] und identi.ca [4] erreichbar.

Auf diesen Seiten kann man, wie auch schon von der freiesMagazin-Webseite gewohnt, Nachrichten und Updates zu verschiedenen Themen verfolgen. Allerdings verwenden wir keine „Gefällt mir“-Buttons, da wir nicht mit den Datenschutzbestimmungen der jeweiligen Dienste einverstanden sind. Es ist uns bekannt, dass es von Heise eine Zwei-Klick-Lösung gibt [5], die wir dennoch nicht einsetzen werden, weil sich am eigentlichen Datenschutz der Unternehmen nichts ändert.

Ende vierter Programmierwettbewerb

Am 30. November 2011 endete die Einsendefrist für die Teilnahme am vierten freiesMagazin-Programmierwettbewerb, wie man bereits auf der freiesMagazin-Webseite lesen konnte [6]. Die Kritik vom vorigen Wettbewerb hatten wir uns zu Herzen genommen und die Aufgabe einfacher gestaltet, was bei Ihnen wohl sehr gut ankam. Es haben erfreulicherweise mehr Teilnehmer ihre Bots eingeschickt als das bei irgendeinem Programmierwettbewerb vorher der Fall war. Insgesamt traten so 20 Bots gegeneinander an.

In den nächsten Tagen findet dann die Ausführung des Wettbewerbs statt, so dass wir die Gewinner noch in der Vorweihnachtszeit auf der freiesMagazin-Webseite und dann in der Januar-Ausgabe im Magazin bekannt geben können.

Index 2011

Wie jedes Jahr gibt es in der letzten Ausgabe des Jahres einen *Jahresindex* auf Seite 63. Der freiesMagazin-Jahresindex 2011 steht auch auf der Webseite im Archiv [7] zum Download bereit.

Und nun wünschen wir Ihnen viel Spaß mit der neuen Ausgabe.

Ihre freiesMagazin-Redaktion

LINKS

[1] https://www.facebook.com/freiesMagazin
[2] https://plus.google.com/u/0/113071049781738007718
[3] https://twitter.com/#!/freiesmaga_open
[4] https://identi.ca/group/freiesmagazin
[5] http://www.heise.de/extras/socialshareprivacy/
[6] http://www.freiesmagazin.de/20111201-vierter-programmierwettbewerb-beendet
[7] http://www.freiesmagazin.de/archiv

Inhalt

Das Editorial kommentieren

Pardus 2011.2 von Hans-Joachim Baader

Obwohl die Linux-Distribution Pardus vom türkischen Staat gefördert wurde, richtet sie sich an ein internationales Publikum. Sie nutzt diverse Eigenentwicklungen, die sie deutlich von anderen abheben. Grund genug, einmal einen näheren Blick zu riskieren.

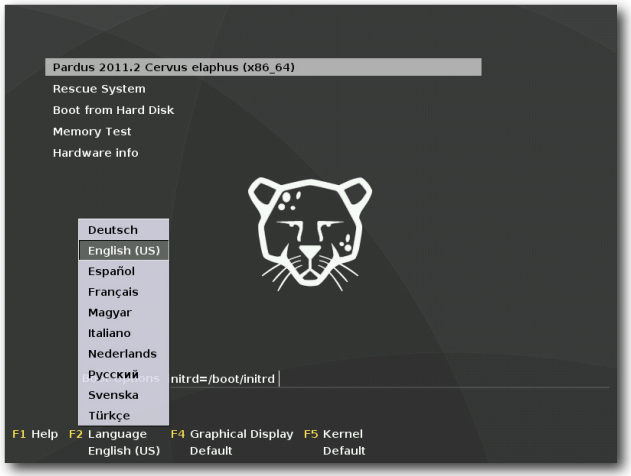
Redaktioneller Hinweis: *Der Artikel „Pardus 2011.2“ erschien erstmals bei Pro-Linux [1].*

Vorwort

Was fällt einem beim Stichwort Pardus ein? Wer sich daran erinnert, dass in diesem Jahr Pardus 2011 sowie die Updates Pardus 2011.1 und Pardus 2011.2 veröffentlicht wurden, dem sind vielleicht Begriffe wie das Installationsprogramm YALI (Yet Another Linux Installer), die eigenständige Paketverwaltung PISI und das Programm Kaptan zur individuellen Anpassung des Desktops (Pardus setzt dabei ganz auf KDE) keine Unbekannten. Durch diese Eigenentwicklungen unterscheidet sich Pardus deutlich von anderen Distributionen wie z. B. Debian, Fedora, Kubuntu, Mageia/Mandriva oder Opensuse, obwohl der größte Teil der Software doch wieder identisch mit den anderen ist. Doch das ist noch nicht alles, wie der Artikel zeigen wird.

Installation

Pardus 2011.2 ist auf die x86-Architektur beschränkt. Es steht in 32- und 64-Bit zur Verfügung und kann frei von der Webseite [2] heruntergeladen werden. Neben Installations-DVDs stehen zum Ausprobieren auch Live-DVDs zur Verfügung. Natürlich ist es auch möglich, mit den DVD-Images einen bootfähigen USB-Stick zu konstruieren.

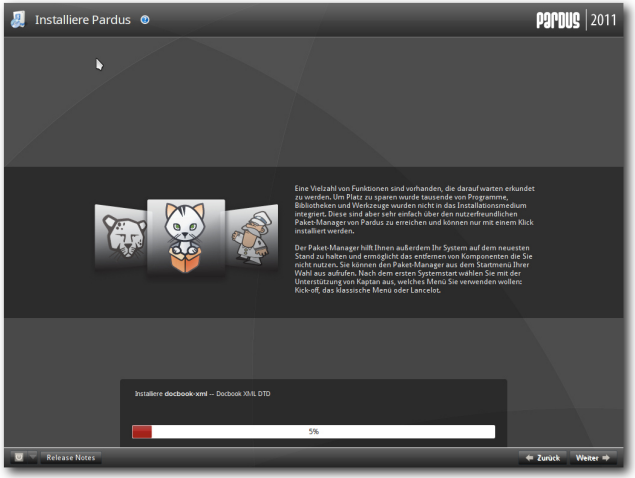


Booten von der DVD.

Die Installation läuft grafisch ab und erinnert deutlich an Fedora. Möglicherweise stammt das Installationsprogramm YALI ja von Anaconda ab, aber über die Hintergründe ist mir nichts bekannt. Schon beim Bootprompt kann man die Sprache wählen und beispielsweise auf Deutsch umschalten. Die deutsche Übersetzung war wohl früher ein Kritikpunkt, ist inzwischen aber bis auf Kleinigkeiten gut.

Die Installation erlaubt zuerst die Auswahl der Tastatureinstellungen, dann die lokale Zeit und Zeitzone, und kommt dann zur Partitionierung. Hier lassen sich, wie inzwischen üblich, die Optionen **`Gesamten Speicher verwenden`**, **`Vorhandenes System verkleinern`**, **`Freien Speicherplatz verwenden`** und **`Manuelle Partitionierung`** wählen.

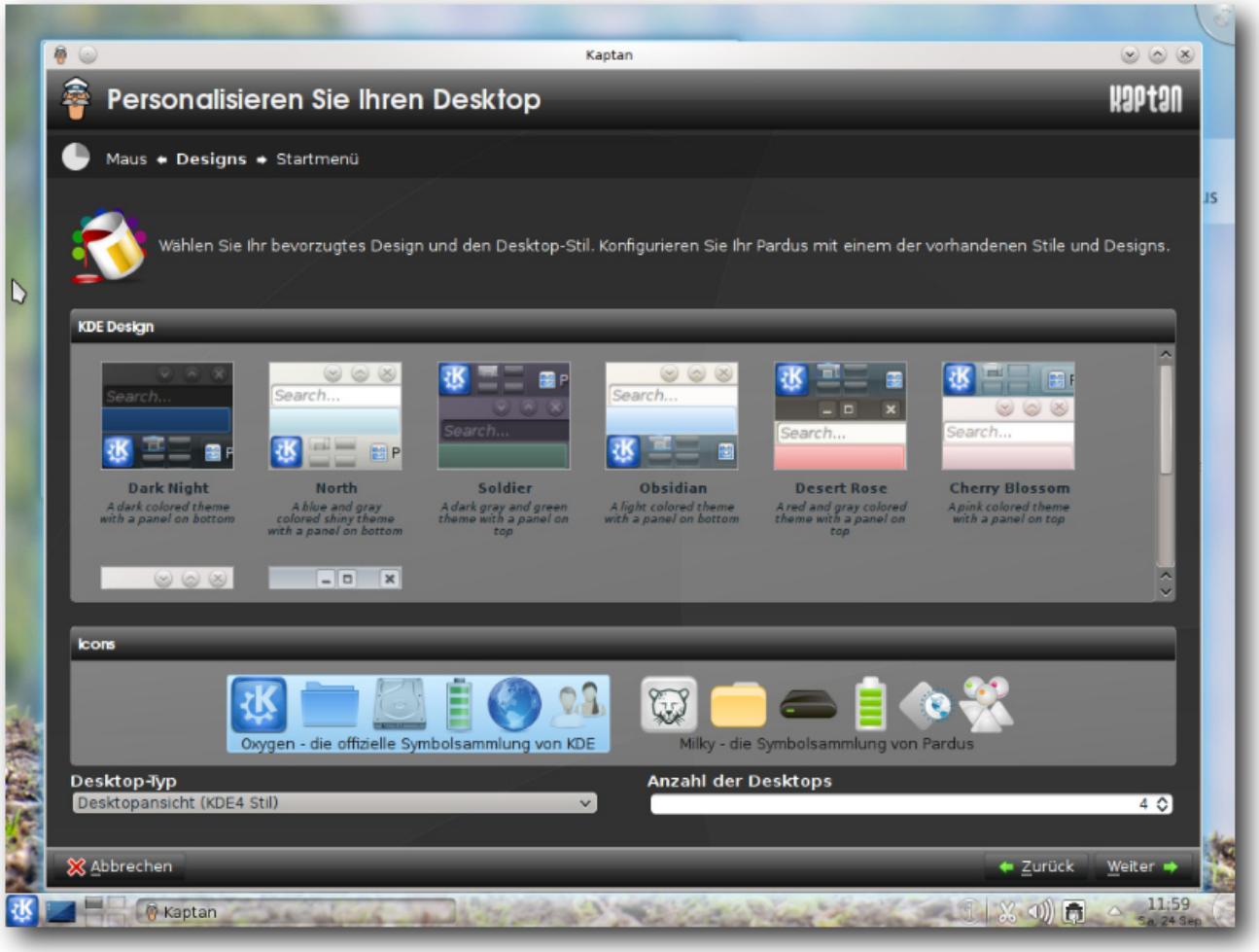
Danach kann man den Bootmanager konfigurieren oder einfach bei den Standardeinstellungen bleiben. Das war es auch schon, da einige weitere Einstellungen erst nach der jetzt beginnenden Installation vorgenommen werden.



Pardus wird installiert.

Nach der erfolgreichen Installation, die eine Weile dauert, wird das System hochgefahren und die weitere Konfiguration vorgenommen. Vor allem erfolgt nun die Festlegung des Root-Passworts und das Anlegen eines Benutzers

für das grafische Log-in. Nun startet KDE zum ersten Mal. Das ist der Auftritt von Kaptan, einem Wizard, der in wenigen Schritten die Anpassung des Desktops ermöglicht. Nach der Einstellung der Maus kann man ein Theme und die Anzahl der virtuellen Desktops einstellen, danach einen Menüstil wählen, dann ein Hintergrundbild und ein Benutzerbild aussuchen, Einstellungen zum Update vornehmen und schließlich, allerdings optional und standardmäßig ausgeschaltet, sein Hardwareprofil an den Distributor senden.



Kaptan hilft beim Einrichten des KDE-Desktops.

Ausstattung

Die Standardinstallation von Pardus 2011.2 bietet etwa 960 Pakete auf. Enthalten sind unter anderem KDE SC 4.6.5 zusammen mit KDE PIM 4.4.11, LibreOffice 3.4.3, Clementine 0.7.1, Digikam 1.9.0, Firefox 5.0, GIMP 2.6.11, NetworkManager 0.8.5.91, GStreamer 0.10.32.4, FFmpeg 0.6.1, PulseAudio 0.6.1, Bash 4.1, OpenSSH 5.6, lcms 1.19, Samba 3.5.10, Python 2.7.1, Perl 5.12.2, Ruby 1.8.7, Lua 5.1.4 Tcl 8.5.10, X-Server 1.9.5 und Kernel 2.6.37.6. Der Standard-Browser ist Firefox, aber Konqueror steht ebenfalls zur Verfügung.

Mehr als 3700 weitere Pakete sind aus den Online-Archiven erhältlich, darunter die Desktopumgebungen Enlightenment, LXDE, Xfce und GNOME 2.32, Entwicklungs- und Server-Werkzeuge sowie Spiele. Darüber hinaus gibt es hunderte von Paketen, die von der Gemeinschaft erstellt wurden und in zusätzlichen Repositorien verfügbar sind. Die Distribution hat damit einen ähnlichen Umfang wie andere, ist in der Standardinstallation recht aktuell und bringt alles mit, um ein weitgehend kompatibles Linux darzustellen.

Betrieb

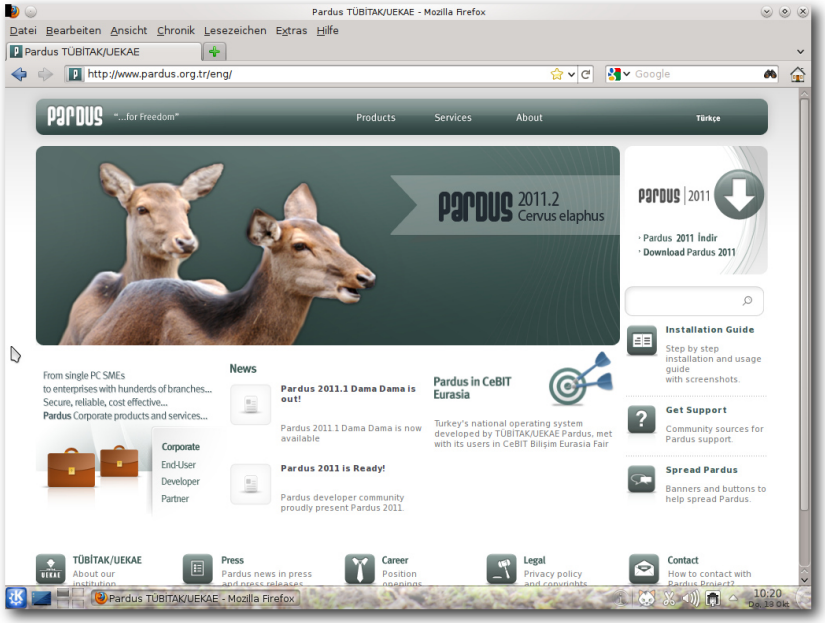
Pardus 2011.2 startet recht flott – 25 Sekunden waren es bis zum Log-in-Fenster. Für eine durchgehende grafische Darstellung wird Plymouth verwendet. Der Speicherverbrauch ist mit über 450 MB nach dem Start recht hoch. Ein Grund dafür ist die in Python geschriebene Paketverwaltung, aber auch Akonadi trägt einiges bei.

Bei längerem Betrieb erhöht sich der Speicherbedarf noch, da sowohl knotify4 als auch die als Icon laufende Paketverwaltung permanent CPU-Leistung und Speicher verbrauchen. Im Laufe eines einzelnen Tages ist der Zuwachs aber noch vernachlässigbar. Die standardmäßig installierte KDE-Oberfläche sieht im Wesentlichen aus wie bei anderen Distributionen.

Will man einen Dienst starten, so sucht man aus Gewohnheit im Verzeichnis **`/etc/init.d`** nach einem entsprechenden Skript. Bei Pardus findet man nur leere Verzeichnisse vor und stellt überrascht fest: Pardus benutzt ein ganz anderes Init-System. Weder das traditionelle SysV-Init noch Upstart noch Systemd steuern den Betrieb bei Pardus. Stattdessen kommt das selbstentwickelte Init-System Mudur (von türkisch Müdür, Direktor) zum Einsatz. Es ist in Python geschrieben, wovon man sich unter **`/sbin/mudur.py`** selbst überzeugen kann. Das Programm wird offenbar von **`/etc/inittab`** gesteuert und weitere Steuerdateien liegen unter **`etc/mudur`**. Die zu startenden Dienste werden dabei durch Dateien definiert, deren Name den Dienst angibt und deren Inhalt keine Rolle zu spielen scheint, weshalb sie leer sind.

Multimedia im Browser und auf dem Desktop

Die Softwarepatente in den USA berühren Pardus als europäische Distribution nicht. Daher sind alle wichtigen freien Codecs im Standardumfang enthalten, gleichgültig ob sie von Patenten belastet sind oder nicht. Dadurch ergibt sich unter Pardus die ungewohnte Situation, dass sich alle relevanten Medienformate ohne Nachinstallation oder Konfiguration abspielen lassen. In der Tat werden Dateien, die man mit dem Dateimanager Dolphin öffnet, immer dem korrekten zuständigen Programm übergeben. In dieser Hinsicht ist Pardus perfekt.

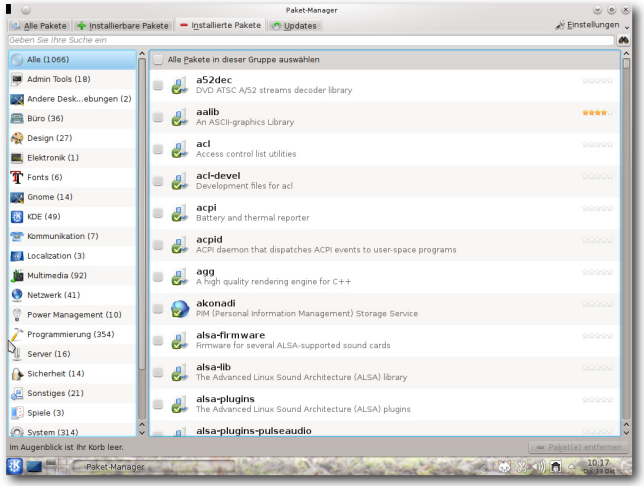


Firefox mit der Pardus-Homepage.

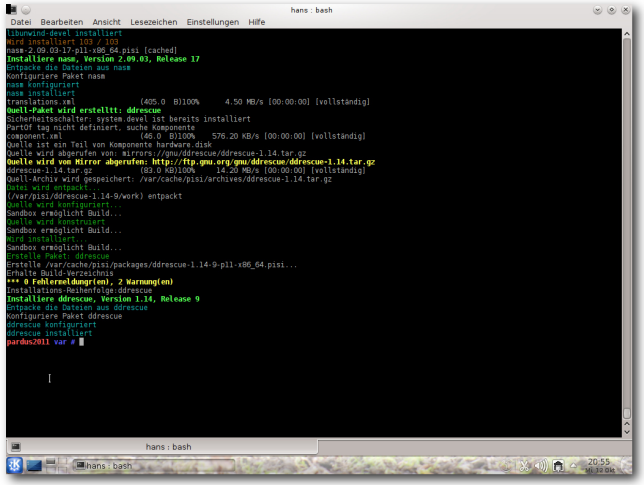
Die Perfektion setzt sich in Firefox fort. Videos von verschiedenen Webseiten funktionierten in allen getesteten Fällen dank der vorinstallierten Plugins, zu denen auch Java und Adobe Flashplayer 11 zählen. In Konqueror scheint der Flashplayer aber leider nicht zu funktionieren, und andere Videos können nur in externen Playern abgespielt werden, was allerdings auch problemlos funktioniert.

Paketverwaltung und Updates

Die mitgelieferte grafische Paketverwaltung, einfach Paket-Manager genannt, kommt einem bekannt vor, erinnert sie doch ein Stück weit an KPackage, Synaptic oder andere entsprechende Programme. Es ist ein Programm, das die Aufgaben Konfiguration, Aktualisierung und Hinzufügen oder Entfernen von Paketen in einem Werkzeug vereinigt. Es fällt zunächst nicht auf, dass im Hintergrund bei Pardus etwas ganz anderes arbeitet als bei Fedora oder Ubuntu. Denn auch die Paketverwaltung ist eine Eigenentwicklung, und auch sie wurde in Python implementiert. Das Hauptwerkzeug der Paketverwaltung heißt Pisi [3], was für „Packages Installed Successfully, as intended“ steht. Es ist weder zu RPM noch zu Debian-Paketen kompatibel, die Gründe dafür sind auf der Pisi-Projektseite nachzulesen. Das Kommandozeilenprogramm „pisi“ erinnert in seiner Syntax an Zypper von openSUSE, enthält aber auch Elemente von Gentoo, denn der Befehl **`pisi em(erge)`** lädt Quellcode-Pakete aus den Repositories herunter, kompiliert und installiert sie. Damit das funktioniert, muss man lediglich das



Die Paketverwaltung.



pisi emerge in Aktion.

passende Quellcode-Depot als Paketquelle hinzufügen. Um alles andere kümmert sich Pisi. Besonders interessant dürfte das sein, um Pakete zu installieren, die für andere Pardus-Versionen konstruiert wurden.

Pisi spielt zusammen mit dem Konfigurationssystem COMAR [4] (eigentlich ÇOMAR), dem es die gesamte Konfiguration überlässt. COMAR, eine weitere Neuentwicklung, steht für Configuration Manager. COMAR arbeitet auch mit dem Init-System Mudur zusammen.

Eine bemerkenswerte Erweiterung von Pisi ist Pisibul [5], das Software aus den Quellen kompiliert und paketiert. Leider ist die letzte Version 0.24 schon vier Jahre alt. Sie lässt sich zwar installieren, funktioniert aber nicht mehr, da sie wohl nicht an KDE 4 angepasst wurde.

Fazit

Pardus 2011.2 ist eine bemerkenswert frische Distribution, die einen Test wert ist. Ihr größter Vorteil ist vielleicht, dass sie erst ab 2003 entstand und nach den ersten Anfängen dazu überging, traditionelle Paradigmen zu überdenken und durch Neuimplementierungen zu ersetzen. Alles, was Pardus von anderen Distributionen unterscheidet, wurde in Python implementiert, was den Entwicklern zufolge zu höherer Geschwindigkeit und besserer Wartbarkeit führt. Zudem sind bestimmte, in anderen Programmiersprachen häufige Fehler in Python ziemlich unwahrscheinlich, und so läuft das System auch sehr stabil und zuverlässig.

Pardus ist nicht die aktuellste Distribution, aber gerade die eher spärlichen Updates und die lange Lebensdauer der einzelnen Versionen werden von vielen Benutzern eher als Vorteil denn als Nachteil gesehen. Nur alle ein bis zwei Jahre erscheint eine ganz neue Version. Der Umfang der Distribution ist, wenn man die Online-Repositorien einbezieht, ausreichend, und zusätzliche Repositorien von Anwendern und Anwendergruppen sowie die Möglichkeit, Pakete aus dem Quellcode zu erstellen, erweitern das Angebot erheblich.

LINKS

[1] http://www.pro-linux.de/artikel/2/1532/pardus-20112.html
[2] http://www.pardus.org.tr/en
[3] http://www.pardus.org.tr/eng/projects/pisi/index.html
[4] http://www.pardus.org.tr/eng/projects/comar/index.html
[5] http://en.pardus-wiki.org/Pisibul

Autoreninformation

Hans-Joachim Baader (Webseite) befasst sich seit 1993 mit Linux. 1994 schloss er sein Informatikstudium erfolgreich ab, machte die Softwareentwicklung zum Beruf und ist einer der Betreiber von Pro-Linux.de.

Diesen Artikel kommentieren

Unity

von Stephan Scholz

Seit Ubuntu 11.04 (Natty Narwhal) ist die Benutzeroberfläche Unity fester Bestandteil des Betriebssystems. War ein Wechsel zu Ubuntu-Classic hier noch möglich, hat sich die Lage nach einer Standardinstallation von Ubuntu 11.10 geändert: Nun kann man (nur noch) zwischen der 3D- und der 2D-Variante von Unity auswählen – es folgt ein Blick auf die Neuerungen.

Der Anmeldebildschirm

Unter dem GNOME Display Manager verhält sich der Anmeldevorgang wie gewohnt. Ab der Ubuntu-Version 11.10 verwendet Ubuntu den Display Manager LightDM.

Mit diesem ist es möglich, die Barrierefreiheit zu gewährleisten, z. B. mittels Bildschirmtastatur oder Bildschirmlupe. Diese Tools sind für Tablet-PCs besonders wichtig, da diese meist über keine Tastatur verfügen.

Der Startbildschirm

Nach dem Anmeldevorgang befindet man sich auf der ersten Arbeitsfläche. Von dort aus gelangt man über die linke Leiste (Starter) in das Unity Menü (Dash), in die vorgegebenen Programme und zum Mülleimer.

Am oberen Rand befinden sich Menüs zum schnellen Zugriff auf das Me-Menü, auf Hardware-Dienste (wie Audio- und Netzwerk-Einstellungen) und andere Dienste eines modernen System-Trays. Eine große Neuerung, die mit Unity eingeführt wurde, ist, dass Programme ihre Menüleiste in diesen oberen Rand legen. Über diesen Schnellzugriff gelangt man beim Darüberfahren mit der Maus, wenn kein Programmfenster aktiv ist, zu den aus GNOME 2.3 bekannten Verknüpfungen zu den „Persönlichen Ordnern“.

Übersicht über die Dash.

Der Starter – Launcher

Der Starter ist ein zentrales Instrument zur Orientierung auf der Unity-Oberfläche. Mit Hilfe des Starters greift man schnell auf Programme, Dateien und das Unity-Menü (Dash) zu. Er wurde für kleine Desktops entwickelt und dient der leichten Übersicht auf der Oberfläche. Anders als bei bekannten Docks ist der Starter schon vollständig vorkonfiguriert, sodass nur die enthaltenen Programme ausgetauscht, entfernt und neue hinzugefügt werden können. Über einen Rechtsklick auf ein Programm im Starter kann der Benutzer bestimmen, ob das Programm geschlossen, gestartet oder im Starter verankert werden soll.

Das Unity-Menü – Dash

Zum Unity-Menü gelangt man über einen Klick auf das Ubuntu-Symbol oberhalb des Starters (ab 11.10 innerhalb desselben). Das Unity-Menü bietet die Möglichkeit, auf Programme, Ordner und Dateien thematisch und alphabetisch sortiert zugreifen zu können. Die Suchfunktion durchsucht Dokumente und Programme, ob diese thematisch, namentlich oder inhaltlich (bei Textdateien) in das Suchraster passen. Die Größe der Dash kann entweder per Shelleingabe (Ubuntu 11.04) oder ab Ubuntu 11.10 über Icons geändert werden.

Programme und Ordner

Die Unity-Oberfläche orientiert sich stark an modernen Oberflächen, ähnlich wie bei den Mac-

OS-X-Betriebssystemen. Wie bereits beschrieben sind die Menüleisten der Anwendungen nicht wie gewohnt im Programmfenster, sondern am oberen Rand zu finden. Eine weitere Neuerung ist, dass die Größe der Programmfenster über das Schieben an den linken, rechten und oberen Rand teilweise oder vollkommen maximiert wird.

Das Software-Center

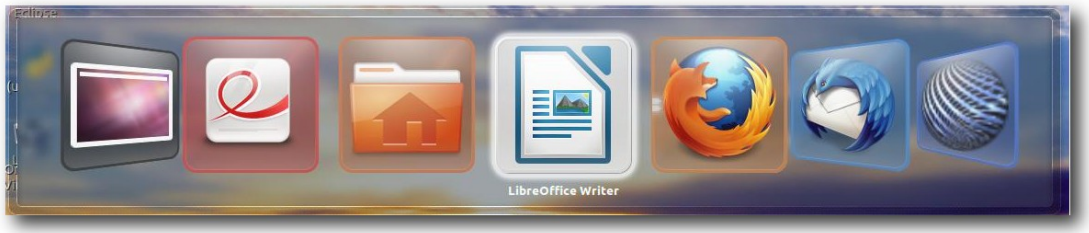
Ab Ubuntu 11.10 wurde der Paketmanager Synaptic komplett durch das Software-Center ersetzt. Das Nachinstallieren von Synaptic ist trotzdem möglich. Das Software-Center bietet vor allem für Einsteiger eine übersichtliche Oberfläche. Die Suche nach geeigneten Programmen wird durch Icons und Programmbeschreibungen deutlich verbessert. Es ist möglich, nach einer Installation das installierte Programm direkt im Launcher als Standard zu integrieren. Über das Tool quickly können nun auch eigene Programme in die Ubuntu-Repositories geladen werden. Weitere Informationen findet man auf der Developer Seite von Ubuntu [1].

Die Systemeinstellungen

Neu ist ab Ubuntu 11.10 die zentrale Verwaltung der Systemeinstellungen. Über diese gelangt man zu den wichtigsten Konfigurationen. Man findet die Systemeinstellungen im Sitzungsmenü in der rechten oberen Ecke.

Weitere Neuerungen

Programme wie Mozilla Firefox, Evolution (Ubuntu 11.04), Thunderbird (Ubuntu 11.10) und Banshee sind in Unity vollständig integriert.



Darstellung des Umschaltens bei Unity-3D.

Durch diese Integration wird beispielsweise eine neue E-Mail im Message-Menü blau dargestellt. Mit Ubuntu 11.10 kann man die Unity-Oberfläche auch auch ohne 3D-Beschleunigung einsetzen. Unity-2D verfügt zwar über die gleichen Tools wie Unity-3D, jedoch wurden die grafischen Effekte, bei denen man eine 3D-Beschleunigung braucht, herausgenommen. So wird etwa der Starter, die Dash und das Umschalten per Alt und Umschalt-Taste klobig dargestellt.

Zusammenfassung

Die Unity-Oberfläche ist, ähnlich wie die GNOME-Shell, ein grafischer Aufsatz auf die GNOME 2.3 und GNOME 3.0 Oberfläche. Außerdem verwendet Unity, um die vielen Grafikeffekte verarbeiten zu können, den Fenstermanager Compiz. Somit macht die Kombination aus GNOME-Oberfläche, Compiz und Canonical als Entwicklerteam Unity bedienungsfreundlich und fit für die Zukunft, besonders um proprietären Betriebssystemen die Stirn zu bieten. Diese Bedienungsfreundlichkeit geht jedoch zu Lasten von Ubuntu/Linux-Nutzern, die die neue Oberfläche als zu umständlich betrachten. Möchte man beispielsweise tiefgreifende Änderungen an Unity vornehmen, muss man Dconf oder CompizConfig benutzen [2]. Außerdem ergeben sich im Detail Fehler, wie das fehlende Hinzufügen von eigens erstellten wechselnden Hintergründen (Slideshow-Wallpapers). Falls man Hilfe, Informationen oder aktuelle Neuigkeiten zu Ubuntu und Unity sucht sind die Internetseiten von „OMG! Ubuntu!“ [3] und „ubuntuusers“ [4] hilfreich.

LINKS

[1] http://developer.ubuntu.com/

[2] http://askubuntu.com/questions/29553/how-can-i-configure-unityubuntuusers.de

[3] http://www.omgubuntu.co.uk/

[4] http://ubuntuusers.de

Autoreninformation

Stephan Scholz arbeitet seit 2010 ausschließlich mit Linux bzw. Ubuntu. Er interessiert sich für innovative Linux-Technologien.

Diesen Artikel kommentieren

Der November im Kernelrückblick

von Mathias Menzer

Basis aller Distributionen ist der Linux-Kernel, der fortwährend weiterentwickelt wird. Welche Geräte in einem halben Jahr unterstützt werden und welche Funktionen neu hinzukommen, erfährt man, wenn man den aktuellen Entwickler-Kernel im Auge behält.

Linux 3.2

Mit der ersten Vorabversion von Linux 3.2 [1] bekam der Kernel auch gleich einen neuen Namen: Auf die „Divemaster Edition“ folgt „Saber-toothed Squirrel“ (in etwa: Säbelzahn-Eichhörnchen). Der Patch, der ein 3.1-Archiv auf 3.2-rc1 befördert, verirrte sich nicht nur in den falschen Pfad, er wurde nicht im Verzeichnis **`/testing`** sondern direkt unter v3.0 [2] veröffentlicht, und war diesmal verhältnismäßig groß. Das lag jedoch an Änderungen der Struktur des Kernel-Quellcodes, in der ganze Bäume verschoben wurden. Während die Versionsverwaltung Git damit problemlos zurecht kommt und das Verschieben innerhalb des Dateibaums abbilden kann, müssen die betroffenen Dateien im Patch einmal gelöscht und am neuen Ort wieder erstellt werden. Diese Aufräumarbeiten dauern schon länger an und sollen die Struktur übersichtlicher machen. Prominent waren das Zusammenziehen der TTY-Umgebung (die Umsetzung der Terminal-Schnittstelle) mit den Character Devices (zeichenorientierten Geräten) (siehe „Der Januar im Kernelrückblick“, **freies**Magazin 02/2011 [3]), diesmal kamen der Bereich der Netzwerktreiber, die ARM-Architektur und der Zweig von User Mode Linux [4] dran. Die bedeutendste Neuerung in Torvalds Augen ging in der Masse dieser Umbauarbeiten unter: Änderungen an der virtuellen Speicherverwaltung [5], die die Steuerung von Schreibvorgängen auf Datenträger verbessern und dadurch für die meisten Endanwender spürbar sein sollen.

Linux 3.2-rc2 [6] konnte wieder im korrekten Pfad gefunden werden, nachdem Torvalds seine Skripte für die Veröffentlichung des Kernels korrigiert hatte. Es wurden insbesondere die Komponenten des freien Nouveau-Treibers für NVIDIA-Grafikchips mit Korrekturen bedacht und Erweiterungen an der Architektur-Unterstützung für Motorolas 6800-Prozessorfamilie vorgenommen. Dazu kommen Ergänzungen der Dokumentation für den DRM-Bereich (Direct Rendering Manager) und das Kernel-Testskript **`ktest.pl`**.

Den -rc3 [7] legte Torvalds dann pünktlich zu Thanksgiving auf. Fiel der -rc2 noch vergleichsweise klein aus, bietet der -rc3 ein eher gewohntes Bild. Die Änderungen verteilen sich einigermaßen gleichmäßig über den ganzen Kernel, der DRM-Bereich sticht mit verschiedenen Korrekturen an den Treibern für Intel- und AMD/ATI-Grafik ein klein wenig hervor. Der WLAN-Treiber iwlwifi sorgte in -rc2 noch für eine Kernel Panic [8], wenn der WLAN-Chip abgeschaltet und der Treiber entladen werden sollte. Eine Änderung der Reihenfolge, in der die einzelnen Komponenten des Systems deaktiviert werden, soll das Problem nun beheben.

Waren die Vorabversionen 2 und 3 mit einer eher steigenden Anzahl an Änderungen behaftet, so deutet Linux 3.2-rc4 [9] auf eine Beruhigung der Entwicklung hin. Entsprechend sind die Änderungen auch überschaubar. An der ARM-Architektur selbst wurde abermals gearbeitet, ebenso an Samsungs auf ARM basierendem System-on-Chip-Plattform Exynos bzw. deren Grafik-Treiber. Auch btrfs wurde mit Fehlerkorrekturen bedacht. Vielleicht legt Torvalds uns dieses Jahr wieder einen Weihnachtskernel unter den Christbaum, da jedoch noch einige Probleme offen sind, lässt sich das nicht mit Sicherheit sagen.

ASPM

Ein langes Leiden begann mit Kernel 2.6.38, als ein Problem mit dem Active State Power Management (ASPM) [10] des PCI-Express-Bus auftrat. Betroffene Notebook-Nutzer mussten fortan mit drastisch verkürzten Laufzeiten leben, da die PCIe-Umgebung sich nicht mehr in den Schlaf schicken ließ.

Ursache war eine Änderung, die eigentlich Probleme in dem Fall verhindern sollte, dass das BIOS die Verwendung von ASPM zu unterbinden sucht. Hierbei wird eine Einstellung ausgelesen, die jedoch in vielen Fällen falsch gesetzt ist und

somit bewirkt, dass ASPM, obwohl eigentlich verfügbar, deaktiviert wird. Die Folge ist ein erhöhter Energiebedarf des PCIe-Busses, da er dauerhaft aktiviert bleibt.

Zwischenzeitlich wurde ein Patch [11] auf Basis des ursprünglichen Übeltäters vorgestellt, der jedoch ASPM nur dann wirklich deaktiviert, wenn die vollständige Kontrolle über PCIe beim Betriebssystem liegt und es damit die eigenen Mechanismen zum Energiemanagement nutzen kann anstelle des in PCIe integrierten ASPM. Das Ubuntu Kernel Team testet den Patch mittels eines angepassten Kernels für Ubuntu 11.10 und die Entwicklungsversion 12.04 [12].

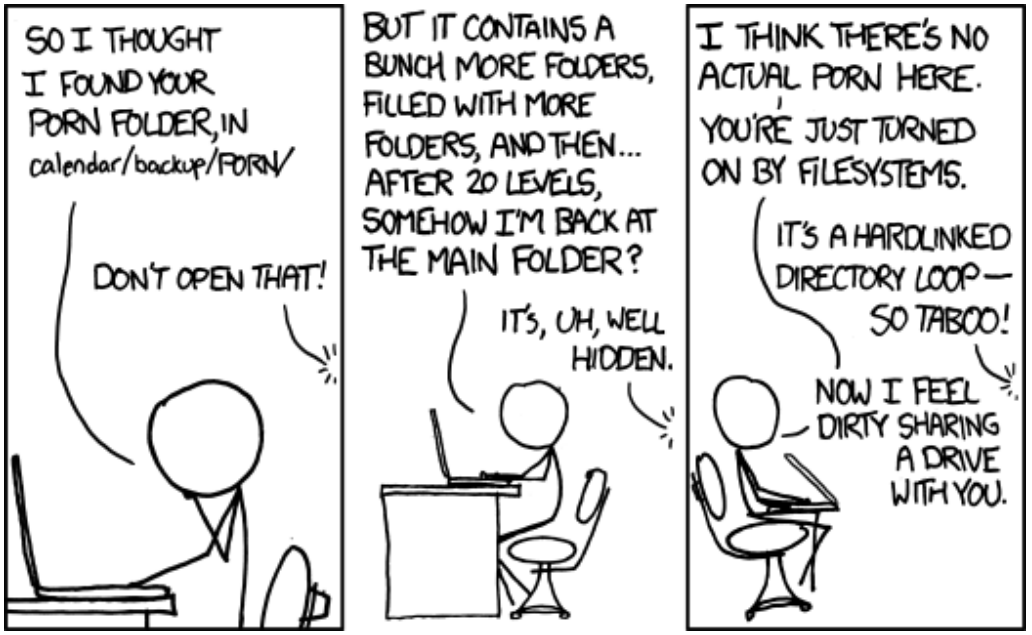
LINKS

[1] https://lkml.org/lkml/2011/11/7/562
[2] https://www.kernel.org/pub/linux/kernel/v3.0/
[3] http://www.freiesmagazin.de/freiesMagazin-2011-02
[4] https://de.wikipedia.org/wiki/User_Mode_Linux
[5] https://de.wikipedia.org/wiki/Virtuelle_Speicherverwaltung
[6] https://lkml.org/lkml/2011/11/15/237
[7] https://lkml.org/lkml/2011/11/23/578
[8] http://de.wikipedia.org/wiki/Kernel_panic
[9] https://lkml.org/lkml/2011/12/1/517
[10] http://en.wikipedia.org/wiki/Active_State_Power_Management
[11] https://lkml.org/lkml/2011/11/10/467
[12] https://wiki.ubuntu.com/Kernel/PowerManagement

Autoreninformation

Mathias Menzer (Webseite) hält einen Blick auf die Entwicklung des Linux-Kernels und erfährt frühzeitig Details über interessante Funktionen.

Diesen Artikel kommentieren



„Porn Folder“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/981

Python-Frameworks für HTML-Formulare

von Jochen Schnelle

Python ist heutzutage eine feste Größe unter den Programmiersprachen – auch, wenn es um das Schreiben von Webanwendungen geht. Wer eine solche programmiert, möchte früher oder später auch Daten vom Nutzer eingeben lassen. Hier kommen dann HTML-Formulare ins Spiel. Im Rahmen dieses Artikels werden fünf verschiedenen Frameworks für Python vorgestellt, welche den Umgang mit diesen Formularen erleichtern sollen.

Warum ein Formular-Framework?

Zugegebenermaßen ist das Erstellen eines Formulars in HTML kein Hexenwerk, sondern vergleichsweise sogar einfach. Ein Framework dient aber nicht alleine zur Erstellung des Formulars aus Python heraus, sondern übernimmt direkt auch noch die Typenkonvertierung und Validierung der Eingaben, was für den Programmierer äußerst praktisch ist.

Gibt man Daten in ein HTML-Formular ein, werden diese vom Browser an den Server bzw. die Anwendung als Text übertragen, sodass in der Anwendung erst einmal alles als String ankommt. Nun gibt es aber eine Vielzahl von Fällen, bei der als Eingabe beispielsweise eine Zahl erwartet wird, wie z. B. beim Alter in Jahren oder einer Hausnummer. Hier führen die Frameworks direkt eine Typenprüfung und -konvertierung von String nach Integer durch – oder melden einen Fehler, wenn der eingegebene Wert keine ganze Zahl ist. Somit kann das unter Umständen recht aufwendige Prüfen und Konvertieren „von Hand“ entfallen. Alle Frameworks stellen solche Felder und Prüfungen auch für Gleitkommazahlen, Datum usw. bereit.

Hat man alle Daten vom Nutzer erhalten und erfolgreich konvertiert, stellen alle Frameworks als weiteren Schritt der Validierung eine Überprüfung der Daten bereit. In deren Rahmen wird getestet, ob z. B. alle Pflichtfelder ausgefüllt wurden. Somit kann auch dafür die händische Prüfung entfallen.

Sollte die Konvertierung oder Validierung fehlschlagen, so erzeugen alle Frameworks direkt eine Liste von Fehlern, zumeist unterteilt in den Feldnamen und den zugehörigen Fehler. Diese kann dann angezeigt werden, zusammen mit der Aufforderung an den Nutzer, das Formular erneut und richtig auszufüllen.

Ein weiterer Vorteil ist, dass die meisten Frameworks automatisch ausklappbare Auswahlfelder generieren können, sodass das lästige Tippen von `<option>...</option>` entfällt.

Client-seitige vs. Server-seitige Validierung

Natürlich können in ein HTML-Formular eingegebene Daten auch Client-seitig via Javascript geprüft werden, diverse Javascript-Frameworks stellen hierfür die benötigten Funktionen bereit. Nichtsdestotrotz sollte immer zusätzlich auch eine serverseitige Validierung erfolgen, da der HTML-Header, welcher die gesendeten Formulardaten enthält, auch nach der clientseitigen Validierung noch manipuliert werden kann. Abgesehen davon ist auf dem Server die Konvertierung der Daten so oder so erforderlich. Und dieser Vorgang ist bei allen vorgestellten Frameworks mit der Validierung verbunden.

Aufbau des Artikels

Im Rahmen dieses Artikels werden fünf verschiedene Frameworks für Python zum Umgang mit HTML-Formularen vorgestellt: Deform, WTForms, FormAlchemy, Fungiform und Flatland. Dabei handelt es sich in der Tat um eine Vorstellung, nicht etwa um einen Vergleichstest oder Ähnliches.

Mit jedem Framework sollen zwei Formulare erstellt werden. Dabei kommen verschiedene Feldtypen und Validatoren zum Einsatz. Das erste Formular heißt `UserData` und enthält drei Felder: Name, Geburtstag und Geschlecht. Der Name darf beliebig sein, außer „admin“ oder „superuser“. Das Feld Geschlecht soll ein Auswahlfeld mit den Auswahlmöglichkeiten „männlich“ und „weiblich“ sein, das Feld Geburtstag soll optional sein. Im zweiten Formular namens `LoginForm` soll ein einfaches Login-Formular erstellt werden. Auch hier gibt es drei Felder: E-Mail-Adresse, Passwort und Wiederholung – alles Pflichtfelder. Die eingegebene E-Mail Adresse soll einer rudimentären Prüfung unterzogen werden, nämlich

ob ein @-Zeichen darin vorkommt. Ein weiterer Validator soll sicherstellen, dass die Eingaben für Passwort und Wiederholung identisch sind.

Danach wird innerhalb einer Python-Shell ein wenig mit den Frameworks und Formularen experimentiert. Dabei wird z. B. testweise ein HTML-Formular aus der Vorlage generiert, Daten übergeben und eine Validierung durchgeführt.

Am Ende des Artikels werden noch einige Framework-unabhängige Hinweise gegeben, wie die Programme innerhalb einer Webanwendung eingebaut und genutzt werden können.

Alle Beispiele sind unter Python 2.6 und Python 2.7 getestet, die getestete Versionsnummer der Frameworks ist jeweils im zugehörigen Abschnitt angegeben.

Gemeinsamkeiten

Es gibt in der Tat ein paar Gemeinsamkeit der Kandidaten. So arbeiten alle fünf Frameworks intern mit Unicode-Daten – was unter Python auch üblich ist. Dies bedeutet aber auch, dass die vom HTML-Formular gesendeten Daten zuerst nach Unicode dekodiert werden müssen, da eine Webseite immer kodierte Daten schickt (z. B. UTF-8, ISO-8815-1 usw.).

Einige der Frameworks haben bereits einen eingebauten Validator zur Prüfung einer E-Mail-Adresse. Dieser ist aber in allen Fällen wirklich minimalistisch, d. h. es wird nur geprüft, ob ein @-Zeichen vorkommt, gegebenenfalls auch noch, ob ein Punkt zur Abgrenzung der Top-Level-Domain vorhanden ist. Wer also ein HTML-Formular mit Eingabe der E-Mail Adresse schreiben möchte und dann auch E-Mails an diese Adresse versenden will sollte auch noch weitergehende Prüfungen implementieren, wie z. B. das Senden einer Bestätigungsmail.

Alle Frameworks stellen HTML-Code nach der Version 4 dar. HTML 5, welches ja auch bei Formularelementen einige Neuerungen bietet, kommt nicht zum Einsatz. Eine Python-3-Version gibt es scheinbar für keines der Frameworks, jedenfalls gibt es keinen offensichtlichen Hinweis

```
# -*- coding: utf-8 -*-

import colander
from deform.widget import SelectWidget, CheckedPasswordWidget

class UserData(colander.MappingSchema):

    def none_off(value):
        if value in ('admin','superuser'):
            return False
        else:
            return True

    name = colander.SchemaNode(colander.String(),
        validator = colander.Function(none_off,
            message = u'nicht erlaubter Benutzername'))
    birthday = colander.SchemaNode(colander.Date(),
        missing = '0000-00-00')
    gender = colander.SchemaNode(colander.String(),
        widget = SelectWidget(values=(
            (u'männlich',u'männlich'),(u'weiblich',u'weiblich'))))

class LoginForm(colander.MappingSchema):
    email = colander.SchemaNode(colander.String(),
        validator = colander.Email(
            msg = u'Dies ist keine gültige E-Mail Adresse'))
    password = colander.SchemaNode(colander.String(),
        widget = CheckedPasswordWidget())
```

Listing 1: deform_forms.py

darauf. Und zu guter Letzt sei noch erwähnt, dass alle fünf Kandidaten unter einer Open-Source-Lizenz stehen.

Deform

Das erste vorgestellt Framework ist Deform [1], welches in Rahmen des Pylons-Projekts entwickelt wird. Deform ist dabei aber komplett unabhängig von Pylons und Pyramid und kann auch in Kombination mit anderen Webframeworks genutzt werden.

Für diesen Artikel wird die aktuelle und stabile Deform-Version 0.9.3 verwendet. Bei der Installation werden drei weitere Abhängigkeiten mit installiert, nämlich Colander [2], Peppercorn [3] und Chameleon [4]. Während der Nutzer mit den letzten beiden Modulen nicht in direkten Kontakt kommt, wird von Colander direkt gebraucht gemacht, nämlich zum Anlegen des Formularschemas.

Sowohl Deform als auch Colander sind vollständig und umfassend dokumentiert.

Klassen für Formulare mit Deform

Die Definition der HTML-Formulare erfolgt, wie auch für die anderen Frameworks, in Klassen. Die beiden Klassen sehen wie im Listing 1 aus.

Als Erstes erfolgen die notwendigen Importe. Dabei wird zuerst Colander importiert, welches, wie oben bereits erwähnt, zur Definition der Struktur dient. Des Weiteren werden noch zwei „Widgets“ importiert, welche später für die Darstellung von bestimmten Formularfeldern verantwortlich sind.

Wie zu sehen ist, ist jedes Formularfeld ein **SchemaNode**, also ein Knoten des Gesamtschemas. Für jeden Knoten ist wiederum ein Datentyp festgelegt, hier in diesen Beispielen wird nur **String** und **Data** verwendet. Bei den Feldern **gender** und **password** wird zusätzlich noch ein Widget angewendet. Im ersten Fall dient dieses dazu, ein Auswahlmenü zu generieren, im zweiten Fall zum Darstellen einer Passworteingabe, inklusive Wiederholung. Gerade das letzte Widget namens **CheckPasswordWidget** ist recht praktisch, da durch dessen Einsatz nur ein Passwortfeld angelegt werden muss. In den anderen Frameworks sind das Passwort und die Wiederholung eigene Felder.

Des Weiteren kommen hier auch Validatoren zum Einsatz. Diese sind Teil der Klasse **colander.SchemaNode** und müssen nicht separat importiert werden. Über **msg = '...'** können innerhalb der Validatoren eigene Fehlermeldungen festgelegt werden. In der Klasse **UserData** kommt weiterhin ein selbst geschriebener Validator zum Einsatz. Dies ist die innerhalb der Klasse definierte Funktion **none_of**, welche prüft, ob der in das Formularfeld eingegebene Benutzername nicht in der Liste der nicht erlaubten Namen enthalten ist. Per Voreinstellung sind alle Felder als Pflichtfelder gekennzeichnet, d. h. es wird eine Eingabe erwartet. Soll ein Feld optional sein, so wie hier das Geburtsdatum, dann ist dass Argument **missing** zu hinterlegen, wie auch hier im Feld **birthday**. Damit wird ein Vorgabewert festgelegt, der die fehlende Eingabe ersetzt.

Deform in der Konsole

Im Folgenden werden einige „Versuche“ mit Deform in einer Python-Shell durchgeführt. Wie bereits erwähnt wurde bis jetzt nur die Struktur der Formulare mit Hilfe von Colander festgelegt. Zum Überführen in eine Klasse für ein Formular muss zuerst noch die Basisklasse von Deform für alle Formulare importiert werden:

```
>>> from deform import Form
```

Dann werden die beiden Klassen aus Listing 1 importiert:

```
>>> from deform_forms import ↷
UserData, LoginForm
```

Jetzt kann eine Formularklasse abgeleitet werden. Dabei können direkt die gewünschte **action** für das Formular sowie eine Senden-Schaltfläche hinzugefügt werden:

```
>>> ud = Form(UserData(), action = ↷
'/foo', buttons = ('submit',))
```

Die Angabe von **action** und **buttons** ist dabei optional. Das HTML-Formular kann nun gerendert werden:

```
>>> ud.render()
u'<form id="deform" action="/foo" ↷
method="POST" [...] <li title="" id↷
="item-deformField1"> <!-- ↷
mapping_item --><label> <class="↷
desc" title="" for="deformField1">↷
Name<span class="req" id="req-↷
deformField1">*</span></label> <↷
```

```
input type="text" name="name" value=↷
="" id="deformField1"/> [...] </↷
form>'
```

Die Ausgabe ist stark gekürzt, weil der HTML-Code sonst den Rahmen des Artikels überschreiten würde. Wie zu sehen ist, generiert Deform direkt das komplette HTML, also inklusive <form>-Tags etc. Auch wenn es im obigen Beispiel nicht zu sehen ist, bindet Deform auch direkt Javascript-Code mit ein. Die passenden Skripte werden bei der Installation des Frameworks mit kopiert und liegen im Installationsverzeichnis von Deform im Unterverzeichnis **/scripts**. Ebenso bringt Deform direkt passende Stylesheets mit, diese liegen im Unterverzeichnis **/css**. Der Einsatz der Skripte und der Stylesheets in der eigenen Applikation ist aber optional, die gerenderten Formulare können auch ohne Weiteres allein genutzt werden.

Als nächstes werden Daten an das Formular übergeben und diese dann validiert. Die Daten erwartet Deform als Tuple von 2er-Tuplen, was übrigens von den meisten Webframeworks standardmäßig auch geliefert wird:

```
>>> mydata = (('name','Otto'),('↷
gender',u'männlich'),('birthday','↷
1977-1-1'))
>>> ud.validate(mydata)
{'gender': u'm\xe4nnlich', '↷
birthday': datetime.date(1977, 1, ↷
1), 'name': u'Otto'}
```

Und noch eine nicht erfolgreiche Validierung:

```
>>> mydata = (('name','admin'),('gender',u'männlich'),
('birthday','1977-1-1'))
>>> ud.validate(mydata)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.6/dist-packages/deform-0.9.3-py2.6.egg/↷
  deform/field.py", line 519, in validate
    raise exception.ValidationFailure(self, cstruct, e)
deform.exception.ValidationFailure
```

Hier geht Deform einen etwas anderen Weg als die anderen Frameworks: Bei erfolgreicher Validierung wird nicht **True** zurückgeliefert, sondern die validierten Daten als Dictionary. Im Falle einer fehlerhaften Validierung wird eine Exception geworfen. Möchte man die Fehlermeldungen sehen, so muss man erneut die **render()**-Methode der Formularklasse aufrufen. Der HTML-Code enthält dann alle Fehlermeldungen:

```
>>> ud.render()
[...]
class="error" id="error-deformField1">nicht erlaubter Benutzername</p>
[...]
```

Da Deform bei Nicht-Validierung immer einen Fehler wirft, muss in einer Applikation natürlich immer ein **try...except**-Block genutzt werden. Ein Beispiel dazu ist in der Dokumentation zu finden [5].

Weitere Funktionen

Neben den hier gezeigt Feldtypen kennt Deform natürlich auch noch eine Reihe weiterer Felder, z. B. für Integerwerte. Außerdem bietet das Framework unter Einsatz von Javascript auch die Möglichkeit der automatischen Vervollständigung von Eingaben in Felder [6].

Durch den Einsatz von Colander für die Definition des Schemas ist es auch ohne weiteres möglich, komplexer und verschachtelte Datenstrukturen darzustellen.

WTForms

Das nächste Framework ist WTForms [7], welches in der Version 0.6.2 vorliegt. Das Framework ist als stabil und für den produktiven Einsatz bereit gekennzeichnet.

WTForms beansprucht für sich selbst, ein „einfaches“ Framework zu sein, mit dem man schnell und mit wenig Aufwand Formulare erstellen kann. Weiterhin ist das Framework so gestaltet, dass sich damit generierte Formularelemente ohne Probleme in Templates einfügen lassen, ohne dass darin größere Änderungen von Nöten sind.

```
# -*- coding: utf-8 -*-

from wtforms import Form, TextField, SelectField, DateField, \
PasswordField, SubmitField, validators

class UserData(Form):
    name = TextField(u'Name',
        [validators.Required(),validators.NoneOf(
            ('admin','superuser'))])
    birthday = DateField(u'Geburtsdatum',[validators.Optional()])
    gender = SelectField(u'Geschlecht',
        choices=[(u'männlich',u'männlich'),
            (u'weiblich',u'weiblich')])
    send = SubmitField(u'Senden')

class LoginForm(Form):
    email = TextField(u'E-Mail Adresse',
        [validators.Email()])
    pass1 = PasswordField(u'Password',
        [validators.Required(), validators.EqualTo('pass2',
            message=u'Passwörter müssen übereinstimmen')])
    pass2 = PasswordField(u'Password Wiederholung')
    send = SubmitField(u'Senden')
```

Listing 2: wtforms_forms.py

Die Dokumentation von WTForms [8] ist vollständig. Der Umfang ist kompakt, verständlich geschrieben und gut strukturiert, sodass diese schnell gelesen ist.

WTForms hat bei der Installation keine Abhängigkeiten, d. h. es werden nur die Dateien des Frameworks selbst installiert.

Formulare mit WTForms

Wie im Listing 2 zu sehen ist, erfolgen alle notwendigen Importe direkt aus der obersten Ebene des Moduls heraus. Importiert werden **`Form`** – welches die Basisklasse für Formulare ist, diverse Feldelemente sowie die enthaltenen Validatoren.

Die Instanzen der Formulare leiten sich von vordefinierten Feldtypen ab, wobei die weiteren Angaben alle optional sind. Nichtsdestotrotz ist es empfehlenswert, zumindest immer ein Label anzugeben, wie im obigen Beispiel auch. Dieses ist später beim Anzeigen des Formular nützlich. Das Label ist dabei das erste Argument. Als zweites Argument kann eine Liste von anzuwendenden Validatoren übergeben werden, dann können weitere optionale Elemente folgen. Im Falle des Felds **`gender`** ist dies z. B. **`choices`**, welches die Optionen für die Auswahlliste enthält. WTForms erwartet die Optionen als Liste von Tupeln, wobei das erste Element des Tupels der Name der Option ist, das zweite die Option, welche in der Liste angezeigt wird. Das erste und zweite Element müssen also nicht gleich sein. Weiterhin ist die „Submit“-Schaltfläche in WTForms ein eigenes Formularelement.

Im obigen Beispiel werden nur Validatoren verwendet, welche WTForms bereits mitbringt. Da es von diesen eine recht umfangreiche Auswahl gibt, ist eine Definition von eigenen Validatoren nicht notwendig – aber grundsätzlich natürlich möglich. Im Feld **`name`** wird mit Hilfe von **`NoneOf`** geprüft, ob die folgenden Begriffe nicht gleich der Eingabe sind. **`birthday`** ist als optional gekennzeichnet, d. h. erfolgt hier keine Eingabe, validiert das Formular trotzdem. Falsche Eingaben führen natürlich trotzdem zu einem Fehler. Der Validator **`EqualTo`** im Feld **`pass1`** prüft, ob die Eingabe identisch mit der im Feld **`pass2`** ist. Außerdem wird hier mit **`message=...`** eine eigene Fehlermeldung hinterlegt, welche die standardmäßig hinterlegte (englischsprachige) Fehlermeldung ersetzt.

WTForms in der Python-Shell

Nach dem obligatorischen Import der Klassen mit den Formularen

```
>>> from wtforms_forms import ↷
UserData, LoginForm
```

wird eine eigene Klasse **ud** von UserData abgeleitet:

```
>>> ud = UserData()
```

Das Rendern der Formularfelder ist dabei denkbar einfach. Dazu wird einfach das darzustellende Feld der Klasse aufgerufen:

```
>>> ud.name()
u'<input id="name" name="name" type↷
="text" value="" />'
>>> ud.send()
u'<input id="send" name="send" type↷
="submit" value="Senden" />'
```

Im Gegensatz zum weiter oben gezeigten Deform – und auch im Gegensatz zu den noch folgenden FormAlchemy und Fungiform – rendert WTForms nicht das ganze Formular, sondern nur die jeweiligen Felder. Die <form>-Tags inklusive **action=...** müssen also manuell gesetzt werden.

Wie bei der Klassendefinition der Formulare bereits erwähnt, können die Label separat gerendert werden:

```
>>> ud.name.label()
u'<label for="name">Name</label>'
```

Der Vorteil ist, dass später so mehr Flexibilität bei der Darstellung in der Webapplikation besteht, dafür aber auch mehr Code-Zeilen nötig sind.

Als Nächstes werden direkt Daten an das Formular übergeben und die Validierung aufgerufen:

```
>>> ud = UserData(name='Otto',birthday='1977-1-1',gender=u'männlich')
>>> ud.validate()
True
```

Da alle Daten im Gültigkeitsbereich liegen, validiert das Formular natürlich. Im folgenden Beispiel wird für **name** ein nicht gültiger Wert übergeben, sodass das Formular nicht validiert:

```
>>> ud = UserData(name='admin',↷
birthday='1977-1-1',gender=u'mä↷
nnlich')
>>> ud.validate()
False
```

Die Fehler können wie folgt aufgerufen werden:

```
>>> ud.errors
{'name': [u"Invalid value, can't be any of: admin, superuser."]}
```

Da keine eigene Fehlermeldung für diese Validierung in der Klasse hinterlegt wurde, gibt WTForms die Standardfehlermeldung aus. Die Fehler stehen übrigens in einer Liste, weil ein Feld durchaus auch mehrere Fehler enthalten kann, je nachdem, wieviele und welche Validatoren hinterlegt sind.

Weiteres zu WTForms

WTForms bietet neben den hier gezeigten Feldern auch alle anderen gängigen Feldtypen. Das Anlegen eigener, neuer Feldtypen ist ebenfalls möglich. Hinweise dazu findet man in der offiziellen Dokumentation.

Des Weiteren gibt es einige offizielle Erweiterungen zu WTForms [9]. Diese erlauben unter anderem die vereinfachte Nutzung von WTForms in Kombination mit Django, SQLAlchemy und der Google-App-Engine.

FormAlchemy

FormAlchemy [10] trägt von allen hier vorgestellten Programmen die höchste Versionsnummer, nämlich 1.4.1 und ist natürlich entsprechend als „stable“ gekennzeichnet. Wie der Name vermuten lässt, gibt es eine Verbindung zu SQLAlchemy [11], einem der populärsten Object-Relational-Mapper für SQL-Datenbanken unter Python. FormAlchemy kann nämlich für SQLAlchemy-Modelle bzw. gemappte Klassen direkt HTML-Formulare erstellen. Dies ist sehr praktisch, wenn man seine Daten ohnehin über diesen ORM abruft und speichert, zumal FormAlchemy Formulareingaben auch direkt wieder in der Datenbank speichern kann.

Aber das Framework kann auch Formulare „konventionell“ erstellen, so wie die anderen hier vorgestellten Programme auch. Dies wird im Rahmen dieses Artikels behandelt. Der einzige Unterschied zur direkten Anbindung an SQLAlchemy ist übrigens, dass man bei „manuellen“

Formularen die Definition händisch vornehmen muss, alle anderen Schritte sind gleich.

Bei der Installation werden drei Abhängigkeiten mit installiert, nämlich WebOb [12], Webhelpers [13], Tempita [14] und SQLAlchemy.

Klassen für das Formular

Bei der händischen Definition von Formularen werden auch in FormAlchemy Klassen verwendet. Im Gegensatz zu den anderen Frameworks sind diese hier aber normale New-Style-Pythonklassen, d. h. es wird keine generische Basisklasse für Formulare zugrunde gelegt. Das Framework kennt natürlich auch verschiedene Datentypen, im FormAlchemy als „types“ bezeichnet. Und es gibt eine Reihe von eingebauten Validatoren, wie z. B. **`required()`**. Das Anlegen eigener Validatoren ist auch möglich. Außerdem kann man in FormAlchemy bei der Definition eines Felds in einem Formular optional den „Renderer“ festlegen, also wie das Feld später dargestellt wird. Natürlich sind hier alle Feldtypen mit sinnvollen Voreinstellungen vorbelegt, trotzdem besteht hier noch bei Bedarf die Flexibilität, auch Spezialfälle abbilden zu können.

Zu Beginn gibt es die obligatorischen Importe, dann folgt die Definition der Klasse **`UserData`**. Wie zu sehen ist, ist jedes Formularelement vom Typ **`Fields`**, die Festlegung von **`type=...`** ist Pflicht, alle folgenden Punkte optional. **`label(...)`** definiert ein Label, **`required()`** legt fest, ob ein Feld ein Pflichtfeld ist, **`validate(...)`** legt die zusätzlichen Validatoren fest.

```
# -*- coding: utf-8 -*-

from formalchemy import Field, types
from formalchemy.validators import email, length, ValidationError

class UserData(object):

    def check_name(value,field):
        if field.value in ['admin','superuser']:
            raise ValidationError(u'Verbotener Benutzername')

    name = Field(type=types.String).label(u'Name').required()\
        .validate(check_name)
    birthday = Field(type=types.Date).label(u'Geburtstag')
    gender = Field(type=types.String).label(u'Geschlecht').dropdown(
        options=[
            (u'männlich', u'männlich'), (u'weiblich',u'weiblich')])

class LoginForm(object):

    def pass2_validator(value, field):
        if field.parent.pass1.value != value:
            raise ValidationError(u'Keine Übereinstimmung.')

    myemail = Field(type=types.String).required().validate(email)
    pass1 = Field(type=types.String).password().required()
    pass2 = Field(type=types.String).password().required()\
        .validate(pass2_validator)
```

Listing 3: FormAlchemy_forms.py

In der Klasse **`UserData`** wird die Funktion **`check_name`** definiert, welche dann im Feld **`name`** als Validator für den Benutzernamen eingesetzt wird.

Die Klasse **`Field`** kennt übrigens auch ein Attribute **`name`**, welches aber beim manuellen Schreiben von Klassen nicht verwendet werden darf, da sonst später ein Fehler beim Anlegen eines FieldSet geworfen wird. FormAlchemy legt den Namen selber fest, wobei dieser identisch ist mit der aus Field angelegten Klasse, im Falle von **`UserData`** also **`name`**, **`birthday`** und **`gender`**.

Interessant ist bei **gender**, dass Auswahlmenüs erst als normales Feld, hier vom Typ String, definiert werden und erst dann festgelegt wird, dass es eine Dropdown-Liste sein soll. Bei Checkboxen und Radiobuttons geschieht dies auf gleiche Art und Weise. Im zweiten Formular **LoginForm** wird über **password()** bestimmt, dass dieses Feld ein Passwortfeld ist. Weiterhin wird in dieser Klasse zuerst eine eigene Funktion **pass2_validator** definiert, welche später bei **pass2** als eigener Validator dient.

Tests in der Python-Konsole

Um FormAlchemy noch näher kennenzulernen wird eine Python-Konsole aufgerufen, um ein bisschen mit den Formularen zu experimentieren. Dazu werden aus **FormAlchemy_forms.py** beide Klassen importiert. Außerdem wird noch **FieldSet** aus FormAlchemy benötigt.

```
>>> from formalchemy_forms import UserData, LoginForm
>>> from formalchemy import FieldSet
```

Mit Hilfe des FieldSet wird nun die eigentliche Klasse erzeugt, mit der das HTML-Formular später generiert wird und mit deren Hilfe auch Formulardaten validiert werden können.

```
>>> ud = FieldSet(UserData)
```

Der einfache Aufruf von **userdata** zeigt die enthaltenen Felder:

```
>>> ud
<FieldSet with ['birthday', 'gender↷
', 'name']>
```

Das Generieren des HTML-Formulars geschieht über den Aufruf von **render()**:

```
>>> ud.render()
u'<div><label class="field_req" for↷
="UserData--name">Name</label><↷
input id="UserData--name" name="↷
UserData--name" type="text" /></div↷
> [...]'
```

Die Länge der Ausgabe ist dadurch bedingt, dass FormAlchemy Datumsfelder per Voreinstellung als Auswahlmenüs darstellt, zumindest für Monat und Tag. Das Jahr ist ein Textfeld mit einer Länge von vier. Das Feld ist aber trotzdem optional, sofern es nicht in der Klasse als **required()** gekennzeichnet wird.

Außerdem ist ein kurzer Schnipsel Javascript enthalten, welcher den Fokus direkt auf das erste Eingabefeld setzt. FormAlchemy verwendet vergleichsweise lange Namen für die einzelnen Felder, welche sich aus dem Formularnamen, zwei Minuszeichen und dem Feldnamen zusammensetzen, also z. B. **UserData--name**.

Die öffnenden und schließenden <form>-Tags werden nicht automatisch generiert, ebenso erzeugt FormAlchemy nicht die „Senden“-Schaltflächen. Dies muss also später in der Applikation händisch erfolgen.

Es ist auch möglich, nur einzelne Felder aus dem Formular rendern zu lassen:

```
>>> ud = FieldSet(UserData)
>>> ud.configure(include=[ud.name])
>>> ud
<FieldSet (configured) with ['name'↷
]>
>>> ud.render()
u'<div><label class="field_req" for↷
="UserData--name">Name</label>[...]
```

Die Liste hinter **include** enthält die Feldnamen, die dargestellt werden sollen. Es ist auch möglich, ein Liste von Feldnamen auszuschließen. Dafür wird **include** einfach durch **exclude** ersetzt.

Um ein Formular zu validieren, müssen zuerst Daten an das Formular übergeben werden. Dies geschieht direkt beim Anlegen des FieldSet:

```
>>> logindata = {'LoginForm--↷
myemail': 'foo@bar.de',
                 'LoginForm--pass2'↷
                 : 'spamegg',
                 'LoginForm--pass1'↷
                 : 'spamegg'}
>>> lf = FieldSet(LoginForm, data=↷
logindata)
>>> lf.data
SimpleMultiDict([('LoginForm--↷
myemail', u'foo@bar.de'), ('↷
LoginForm--pass2', u'spamegg'), ('↷
LoginForm--pass1', u'spamegg')])
>>> lf.data['LoginForm--pass1']
u'spamegg'
>>> lf.validate()
True
```

Als Nächstes werden Daten übergeben, welche nicht validieren:

```
>>> baddata = {'LoginForm--myemail': 'foo_bar.de',
               'LoginForm--pass2': 'spam',
               'LoginForm--pass1': 'spamegg'}
>>> badform = FieldSet(LoginForm, data=baddata)
>>> badform.validate()
False
>>> badform.errors
{AttributeField(myemail): ['Missing @ sign'], AttributeField(pass2): ['↷
Passw\xc3\xb6rter stimmen nicht \xc3\xbcberein.']}
>>> badform.render()
u'\n\n<div>\n  <label class="field_req" for="LoginForm--myemail"> [...] <↷
span class="field_error">Missing @ sign</span>\n</div>'
```

Auch diese Ausgabe ist gekürzt. Wie nicht anders zu erwarten, gibt der Aufruf von **`validate()`** **`False`** zurück. Die Fehler sind in **`errors`** hinterlegt. Wird das Formular gerendert, so werden die Fehler auch direkt im Quelltext angezeigt.

Weitere Funktionen

Wie in der Einleitung bereits erwähnt, ist der Ursprung von FormAlchemy das Generieren von Formularen aus gemappten Klassen aus SQLAlchemy. Außerdem bietet das Framework auch Klassen und Funktionen, um komplexere Formulare direkt in Form einer Tabelle darstellen zu lassen. Des Weiteren gibt es unter anderem eine Anbindung von FormAlchemy an CouchDB, Pylons sowie Zope, AJAX-basierte Formulare, die auf jQuery zurückgreifen [15] sowie eine Anbindung an Pyramid [16]. Wer sich näher mit FormAlchemy beschäftigen möchte, der sollte einen Blick in die sehr ausführliche und gut strukturierte Dokumentation [17] werfen.

Fungiform

Fungiform [18] entstammt dem Pocoo-Umfeld [19] und war ursprünglich Teil der in Python geschriebenen Blogsoftware „Zine“ [20], wurde aber 2010 als eigenständiges Framework ausgelagert.

Fungiform trägt zwar „nur“ die Versionsnummer 0.1, ist in sich aber ziemlich komplett, weiterhin gibt es scheinbar keine größeren Bugs mehr. Jedenfalls traten bei den Tests mit Fungiform im Rahmen dieses Artikels keine Probleme auf. Apropos Probleme: Es gibt zwei andere „Probleme“ mit Fungiform. Das eine ist, dass es scheinbar keine aktiven Entwickler gibt. Nach der Auslagerung von Fungiform im Sommer 2010 gab es zwar diverse kleinere Korrekturen, aber keine eigentliche Weiterentwicklung. Der letzte Commit zum Quellcode stammt von Ende 2010. Das zweite, nicht ganz so große Problem ist, dass es keine separate Dokumentation im HTML-Format oder ähnlichem gibt. Dies lässt sich aber verschmerzen, da der Quellcode sehr gut und sehr ausführlich dokumentiert ist. Wer sich mit Fungiform beschäftigen möchte, der sollte Gebrauch von der in Python eingebauten help()-Funktion machen oder die Doc-Strings direkt im Quelltext lesen. Die wichtigste Datei ist dabei **`forms.py`** [21], welche so gut wie alle Klassen und Funktionen enthält, die man später auch für das eigene Programm benötigt.

Definition der Klassen

Wie bei den anderen bisher vorgestellten Frameworks wird auch hier die Struktur für das HTML-Formular in einer Python-Klasse hinterlegt. Es gibt die üblichen Feldtypen für Text, Zahlen, Auswahllisten usw. Weiterhin gibt es einige „spezielle“ Formularelemente, wie z. B. zur Eingabe von Komma-separierten oder Zeilenumbruch-separierten Daten. Außerdem kann man auch in Fungiform Validatoren hinterlegen. Was sich bei Fungiform etwas unterscheidet ist, dass einige „Basis-Validatoren“ bereits in den Feldklassen hinterlegt sind. So kann man für einige Klassen wie z. B. **`TextField`** die Attribute **`required`**, **`min_length`** oder **`max_length`** direkt definieren, ohne dafür einen separaten Validator anlegen zu müssen. Davon wird auch im folgenden Beispiel Gebrauch gemacht. Im Gegenzug kennt Fungiform allerdings keine anderen vordefinierten Validatoren – wie z. B. WTForms und Deform – welche unabhängig von einer Klasse sind. Dafür kann man aber sehr einfach eigene Validatoren schreiben.

```
# -*- coding: utf-8 -*-

from fungiform.forms import FormBase, TextField, DateField, \
ChoiceField, PasswordField, ValidationError

class UserData(FormBase):

    def is_valid_name(self,value):
        if value in ['admin','superuser']:
            message = u'unerlaubter Benutzername'
            raise ValidationError(message)

    name = TextField(label=u'Name', required=True,
        validators=[is_valid_name])
    birthday = DateField(label=u'Geburtsdatum')
    gender = ChoiceField(label=u'Geschlecht',
        choices=[u'männlich',u'weiblich'])


class LoginForm(FormBase):

    def is_valid_email(self,value):
        if '@' not in value:
            message = u'ungültige E-Mail Adresse'
            raise ValidationError()

    email = TextField(label=u'Email', required=True,
        min_length=6, max_length=120)
    pass1 = PasswordField(label='Passwort', required=True)
    pass2 = PasswordField(label='Wiederholung', required=True)

    def context_validate(self, data):
        if data['pass1'] != data['pass2']:
            message = u'Passwörter stimmen nicht überein'
            raise ValidationError(message)
```

Listing 4: fungiform_form.py

Wie man sieht, kommen alle Importe aus **fungiform.forms**. Importiert wird die Basisklasse für alle Formulare **FormBase** sowie die benötigten Klassen für Formularfelder und der **ValidationError**, welcher für die selbstgeschriebenen Validatoren benötigt wird.

Die beiden Klassen **UserData** und **LoginForm** besitzen jeweils einen selbstdefinierten Validator. Diesem muss immer **value** übergeben werden, was der zu prüfende Wert ist. Der anschließende Code kann beliebiger Python-Code sein. Wenn die Validierung nicht erfolgreich war, wird der **ValidationError** geworfen, wobei diesem eine benutzdefinierte Fehlermeldung mitgegeben werden kann.

Der sonstige Code der Klassen sollte weitestgehend selbsterklärend sein. Ist ein Feld ein Pflichfeld (wie z. B. **name**), so setzt man das Attribute **required=True**. Übergibt man später Fungiform für die Validierung des gesamten Formulars keinen Wert für dieses Feld, so wird ein entsprechender Fehler geworfen.

In der Klasse **LoginForm** wird ein zusätzlicher Validator namens **context_validate** benutzt, welcher die beiden Passworteingaben vergleicht. Besitzt eine Klasse diesen, so wird er auf die gesamte Klasse (das gesamte Formular) angewendet und nicht, wie die anderen Validatoren, nur auf einzelne Instanzen einer Klasse. Im Gegensatz zu einem „Einzelvalidator“ muss man **context_validate** immmer **data** übergeben, welches alle Formulardaten enthält.

Formulare benutzen

Als Erstes soll wieder der HTML-Quelltext für das Formular generiert werden. Dies geschieht bei Fungiform in zwei Schritten: Zuerst ruft man die Funktion **as_widget()** auf, dann wird erst das eigentliche HTML erzeugt. Als Parameter wird dem Formular dabei die gewünschte Ziel-URL als **action** mitgegeben:

```
>>> from fungiform_form import UserData, LoginForm
>>> ud = UserData(action='/foo')
>>> ud_widget = ud.as_widget()
>>> ud_widget()
u'<form action="/foo" method="post"><dl class="mapping"><dt><label for="f_name">Name</label></dt><dd><input type="text" id="f_name" value="" name="name"></dd> [...] <div class="actions"><input type="submit" value="Submit"></div></form>'
```

Um den Umfang des Artikels nicht zu sprengen, ist auch diese Ausgabe gekürzt.

Ein direktes Rendern ist auch möglich:

```
>>>  ud.as_widget().render()
u'<form action="/foo" method="post"> [...]'
```

Wie zu sehen ist, fügt Fungiform auch die **<form>...</form>** Tags automatisch hinzu, ebenso wird die „Submit“-Schaltfläche automatisch generiert. Wie auch bei den anderen Frameworks wird in den Tags eine **id** angelegt. Hier werden Default-Werte seitens Fungiform verwendet, diese können vor der Formulargenerierung aber auch überschreiben werden.

Die voreingestellte Methode zum Senden des Formulars ist **post**, dies kann aber recht einfach geändert werden:

```
>>> ud.default_method = 'get'
>>> ud_widget = myform.as_widget()
>>> ud_widget()
u'<form action="/foo" method="get">...'
```

Die Label für die einzelnen Felder sind wie folgt abrufbar:

```
>>> ud.name.label
u'Name'
```

Wie die anderen Frameworks kann Fungiform die für ein Formular übermittelten Daten automatisch validieren. Dies kann ebenfalls in der Pythonkonsole getestet werden:

```
>>> mydata = {'name':'Otto','gender':u'männlich','birthday':'1977-7-7'}
>>> ud.validate(mydata)
True
```

Da alle Werte gültig sind, validiert das Formular ohne Fehler. Wie zu sehen ist, wurde auch hier das Datum als String übergeben. Fungiform wandelt diesen automatisch bei der Validierung in ein Datetime-Objekt um. Dies ist zu sehen, wenn der übermittelte Wert aufgerufen wird:

```
>>> ud['birthday']
datetime.date(1977, 7, 7)
```

Alle Daten können wie folgt abgerufen werden:

```
>>> ud.data
{'gender':u'm\xe4nnlich','birthday':datetime.date(1977,7,7),'name':u'Otto'}
```

Als Nächstes werden an das Formular Daten übermittelt, welche nicht gültig sind:

```
>>> mydata = {'name':'admin','gender':u'mann','birthday':''}
>>> ud.validate(mydata)
False
>>> ud.errors
{'name': [u'unerlaubter Benutzername'],
'gender': [u'Please enter a valid choice.']}
```

Wie erwartet validiert das Formular nicht. Die Fehlermeldungen werden in **errors** gespeichert. Ruf man diese auf, sind die Fehlermeldungen als Dictionary zu sehen. Sofern keine eigene Fehlermeldung vorgegeben wurde – wie im Falle des Felds **gender** – wird die Standardfehlermeldung für diesen Feldtyp verwendet.

Der Status der letzten Validierung kann auch aufgerufen werden, ohne das ganze Formular erneut zu validieren:

```
>>> ud.is_valid
False
```

Weitere Funktionen

Fungiform bietet natürlich noch weitere Möglichkeiten als die hier gezeigten. So ist es z. B. auch möglich, eigene Feldtypen zu definieren, Feldern „Widgets“ mitzugeben, um deren Verhalten zu ändern usw. Dies ist auch in der eingebauten Dokumentation (kurz) erklärt.

Eingangs wurde bereits erwähnt, dass auch Fungiform alle gängigen Feldtypen generieren kann sowie ein paar „Extrafelder“ an Bord hat. Des Weiteren unterstützt Fungiform den Einsatz von Recaptchas [22] und bietet mit Bordmitteln Schutz vor Cross-Site-Request-Forgery [23]. Aufgrund der fehlenden Dokumentation müssen auch hier der Quelltext beziehungsweise die Kommentare darin gelesen werden, um herauszufinden, wie diese Funktionen zu nutzen sind.

Flatland

Flatland trägt ebenso wie Fungiform eine recht niedrige Versionsnummer, nämlich 0.0.1, und wird auf PyPi [24] mit dem Status „beta“ deklariert. Laut Homepage des Projekts [25] ist das Framework komplett, frei von größeren Bugs, einsatzfähig und auch tatsächlich im Einsatz. Bei den Tests mit Flatland für diesen Artikel traten in der Tat keine Probleme auf – aber es fehlt noch Dokumentation [26]. Dies ist aktuell in der Tat das größte Manko von Flatland. Es gibt zwar eine schon recht umfangreiche Dokumentation [27], diese beschreibt aber fast ausschließlich das Anlegen von eigenen Klassen und den Einsatz der

```
# -*- coding: utf-8 -*-

from flatland import Form, String, Date
from flatland.validation import Present, ValueIn, IsEmail, \
    ValuesEqual, IsEmail

GENDER_DATA = [u'männlich',u'weiblich']

def not_allowed_name(element, state):
    if element.value in ['admin','superuser']:
        element.errors.append(u'Verbotener Name')
        return False

class UserData(Form):
    name = String.using(label = u'Name',
        validators=[Present(missing=u'Kein Name eingegeben'),
                    not_allowed_name])
    birthday = Date.using(label = u'Geburtsdatum',optional = True)
    gender = String.using(label = u'Geschlecht',
        validators=[Present(missing=u'Geschlechtsangabe fehlt'),
                ValueIn(GENDER_DATA,fail =
                  u'"%(value)s" ist ungültiger Wert für %(label)s.')])

class LoginForm(Form):
    email = String.using(label='E-Mail',
        validators=[
            Present(missing=u'Kein E-Mail Adresse eingegeben'),
            IsEmail(invalid=u'Keine gültige E-Mail Adresse')])
    pass1 = String.using(label = u'Passwort',
        validators=[Present(missing = 'Kein Passwort eingegeben')])
    pass2 = String.using(label = u'Passwort Wiederholung',
        validators=[Present(missing = 'Passwortwiederholung fehlt')])

    validators = [
        ValuesEqual(pass1,pass2)]
```

Listing 5: flatland_forms.py

verschiedenen existierenden Felder. Eine Erklärung, wie man aus einer Formularklasse HTML erzeugt, fehlt z. B. noch vollständig. Glücklicherweise wurde auf der Europython 2010 ein Vortrag zu Flatland gehalten, welcher die Nutzung komplett, wenn auch relativ kurz, erläutert [28]. Wer sich näher mit Flatland beschäftigen möchte, der sollte sich die Folien unbedingt ansehen.

Ähnlich wie das im Rahmen von Deform angesprochene Colander, kann Flatland auch komplexere Datenstrukturen darstellen, da sich mehrere Felder beliebig in Dictionaries und Listen zusammenfassen lassen. Diese dürfen auch wiederum Dictonaries und Listen enthalten. Darauf wird im Rahmen dieses Artikels aber nicht eingegangen.

Anlegen der Formulare

Als Erstes werden wieder die notwendigen Importe durchgeführt, einmal für die verwendeten Felder und einmal für die eingesetzten Validatoren. Ansonsten wird für jedes Feld zuerst der Datentyp festgelegt und danach das Label sowie die anzuwendenden Validatoren.

Weiterhin wird in der Klasse **UserData** ein eigener Validator namens **not_allowed_name** definiert. Dieser wird später im Feld **name** eingesetzt und prüft, ob der eingegebene Name innerhalb der nicht erlaubten Liste von Namen liegt.

Bei der Klasse **LoginForm** wird ein Validator außerhalb der Felddefinition eingesetzt. Dieser wird also auf das gesamte Formular angewendet. Dies ist immer dann praktisch, wenn – so wie hier – Eingaben aus verschiedenen Feldern miteinander verglichen werden sollen. Etwas umständlich ist die Tatsache, dass Flatland zwar selbst definierbare Fehlermeldungen bei nicht erfolgreicher Validierung erlaubt, die Definition an sich aber nicht einheitlich ist. Wie auch im Listing 5 oben zu sehen ist, wird die eigene Fehlermeldung im Falle des Validators **Present** über **missing = '...'** festgelegt, im Falle des Validators **IsEmail** aber über **invalid = '...'**. Einheitlicher ist eher der Weg, den die anderen Frameworks wählen,.

Im Gegensatz zu allen anderen hier vorgestellten Formular-Frameworks kennt Flatland scheinbar keinen eigenen Feldtyp für ein Auswahlfeld. „Scheinbar“ deshalb, weil im Rahmen der Recherche dieses Artikels dieser Feldtyp zwar nicht aufzufinden war, es aber nicht auszuschließen ist, dass das Auswahlfeld schlichtweg noch nicht dokumentiert ist und somit nicht „gefunden“ wurde.

Tests mit Flatland in der Python-Shell

Im Folgenden werden die mit Flatland definierten Klassen für die Formulare ein wenig in der Python-Shell erkundet. Zuerst werden die beiden Formularklassen importiert,

```
>>> from flatland_forms import ↷
LoginForm, UserData
```

und eine eigene Klasse daraus instanziiert:

```
>>> ud = UserData()
```

Der Aufruf von **ud** gibt die Feldnamen und die aktuellen Werte zurück:

```
>>> ud
{u'birthday': <Date u'birthday';↷
value=None>, u'name': <String u'↷
name'; value=None>, u'sex': <String↷
 u'sex'; value=None>}
```

Da noch keine Werte übergeben wurden, ist **value** natürlich überall **None**. Einzelne Felder sind wie bei Dictionaries aufrufbar:

```
>>> ud['name']
<String u'name'; value=None>
```

Die Label für Felder können wie folgt aufgerufen werden:

```
>>> ud['name'].label
u'Name'
```

Als Nächstes werden Daten an das Formular übergeben und validiert:

```
>>> from datetime import date
>>> mydata = {'name':'Otto',
'gender':u'männlich',
'birthday':date(1977,1,1)}
>>> ud.set(mydata)
True
>>> ud.validate()
True
>>> ud
{u'gender': <String u'gender'; ↷
value=u'm\xe4nnlich'>, u'birthday':↷
 <Date u'birthday'; value=datetime.↷
date(1977, 1, 1)>, u'name': <String↷
 u'name'; value=u'Otto'>}
```

Flatland erwartet, wie zu sehen ist, für das Geburtsdatum explizit ein Python-Date-Objekt. Ein String in der Form **1977-1-1**, wie ihn die anderen Frameworks als gültiges Datum akzeptieren, führt in Flatland zu einem Fehler.

Als Nächstes wird das Formular gegen einen ungültigen Datensatz validiert:

```
>>> mydata = {'name':'admin',
'gender':u'männlich',
'birthday':date(1977,1,1)}
>>> ud.set(mydata)
True
>>> ud.validate()
False
```

Wie nicht anders zu erwarten, schlägt die Validierung fehl. Leider speichert Flatland keine Fehlerliste, welche für das ganze Formular gilt. Vielmehr werden die Fehler in den jeweiligen zugehörigen Feldern hinterlegt. Von daher muss man über diese iterieren, um die Fehler zu sehen:

```
>>> for k in ud.iterkeys():
...     print k, ud[k].valid
...     print k, ud[k].errors
...
gender True
gender []
birthday True
birthday []
name False
name [u'Verbotener Name']
```

Flatland ist das einzige der hier vorgestellten Frameworks, das für das Rendern der Formulare nach HTML einen eigenen Import benötigt, nämlich den **Generator**:

```
>>> from flatland.out.markup import↷
 Generator
```

Dies ist übrigens zum Zeitpunkt des Schreibens dieses Artikels vollständig undokumentiert, d. h. in der offiziellen Dokumentation an keiner Stelle erwähnt.

Danach können die Formularfelder generiert werden:

```
>>> gen = Generator()
>>> gen.input(ud['name'], type='↷
text')
<input type="text" name="name" ↷
value="" />
```

Ähnlich wie WTForms rendert Flatland die Felder einzeln. Das komplette Rendern eines ganzen Formulars wird nicht unterstützt. Wie zu sehen ist, wird über **gen.input()** ein <input>-Feld generiert, **type='text'** legt den Typ als Textfeld fest.

Da Flatland scheinbar keine Auswahlfelder unterstützt, müssen diese von Hand „gebaut“ werden:

```
>>> from flatland_forms import ↷
GENDER_DATA
>>> gen.select.open()
u'<select>'
>>> for i in GENDER_DATA:
...     gen.option.open(),\
...     i,gen.option.close()
...
u'<option> männlich </option>'
u'<option> weiblich </option>'
>>> gen.select.close()
u'</select>'
```

Der Generator aus Flatland ist also recht flexibel und kann alle für Formulare benötigten HTML-Tags generieren. Nichtsdestotrotz ist diese Vorgehensweise vergleichsweise aufwändig.

Sonstiges

Neben diversen weiteren Feldtypen sowie der Möglichkeit der Verschachtelung von Feldern kennt Flatland auch noch „Signals“ [29]. Diese können in den Programmcode eingebaut werden und senden während der Verarbeitung von Formularen zu oder nach einem bestimmten Ergebnis, wie einer erfolgreichen Validierung, ein Signal an eine Liste vordefinierter Empfänger. Des Weiteren gibt es eine Anbindung an die beiden Template-Engines „Genshi“ und „Jinja2“ [30].

Weitere Alternativen

Die hier vorgestellten Programme stellen natürlich nur einen Ausschnitt der existierenden Frameworks dar. Ein weiteres ist z. B. FormEncode [31]. Noch mehr Alternativen sind auf PyPi zu finden, wenn z. B. als Suchbegriff „html form“ eingegeben wird.

Wird das für Webapplikationen auch sehr populäre Django [32] genutzt, so steht hier ein eigenes Modul für HTML-Formulare bereit [33], welches natürlich komplett in Django integriert ist.

HTML-Framework in Webapplikationen

Im Rahmen dieses Artikels wurden die verschiedenen Frameworks ausschließlich in der Python-Shell erkundet. Die Nutzung innerhalb einer realen Webapplikation ist aber auch nicht weiter schwierig, zumal das prinzipielle Vorgehen grundsätzlich für alle vorgestellte Frameworks gleich ist.

Es gibt grundsätzlich drei verschiedene Zustände, die im Programmablauf berücksichtigt werden müssen:

- es wurde noch kein Formular generiert
- es wurde ein ausgefülltes Formular von der Webseite gesendet, dieses validiert aber nicht, d. h. es enthält Fehler
- es wurde ein ausgefülltes Formular von der Webseite gesendet und dieses validiert

Weiterhin gilt es zu bedenken, dass alle Frameworks, wie weiter oben bereits erwähnt, Unicode-Daten erwarten, die Webseite aber codierte Daten liefert.

Im Folgenden wird eine minimale Webapplikation mit nur einem Formular mit Hilfe von WTForms und Bottle (siehe Artikel in freiesMagazin 2/2011 [34]) gezeigt. Der Code ist nicht weiter spezifisch für Bottle und sollte sich sehr einfach auch auf andere Webframeworks übertragen lassen. Der Austausch von WTForms gegen ein anderes Formular-Framework sollte sich auch einfach bewerkstelligen lassen. Der Code besteht dabei aus zwei Teilen: Der Applikation an sich und dem zugehörigen Template.

my_form_app.py

template.tpl

In **my_form_app.py** wird zuerst die notwendige Dekodierung von UTF-8 nach Unicode für alle eingegebenen Werte durchgeführt. Dabei ist zu beachten, dass es sich um eine „minimale“ Konvertierung im Rahmen dieses Beispiels handelt. Das im Template generierte HTML stellt nicht sicher, dass die Eingabe wirklich immer UTF-8 ist und fängt auch nicht das mögliche Fehlverhalten älterer Browser (wie z. B. Microsofts Internet Explorer 6) ab.

Da die Namen aller Formularfelder ausschließlich aus ASCII-Zeichen bestehen, ist eine Dekodierung der Feldnamen nicht notwendig. Danach wird die oben beschriebene Prüfung auf den Zustand durchgeführt, wobei dieser für „noch kein Formular generiert“ und „validiert nicht“ praktischerweise zusammengefasst ist.

Das zugehörige Template in **template.tpl** gibt standardmäßig nur das Formular aus. Die Fehler werden nur angezeigt, wenn **errors** vorhanden ist, was nach einer nicht erfolgten Validierung der Fall ist.

Deform kommt übrigens nicht mit dem verwendeten MultiDict von Bottle zurecht, es wird dann ein **ValueError** geworfen. Soll das Beispiel auf Deform übertragen werden, so sollte **req_unicode** als Liste angelegt und dann Feldname und Wert als Tuple darin hinterlegt werden.

Zusammenfassung

Trotz einiger Gemeinsamkeiten und dem gleichen Ziel zeigen sich doch eine Reihe von Unterschieden in den fünf in diesem Artikel vorgestellten Frameworks.

Deform setzt auf ein separates Modul beim Anlegen der Klassen für die Struktur und Felder der Formulare. Beim Rendern wird direkt Javascript mit eingebunden, auch wenn die Nutzung letztendlich nur optional ist.

WTForms, FormAlchemy und Fungiform sind recht ähnlich, was die Definition der Formulare angeht. WTForms erlaubt eine unkomplizierte Definition der Formularfelder und bringt bereits eine Vielzahl von Validatoren mit. Felder werden einzeln gerendert, was die Integration in eine Applikation recht einfach macht – auch im Nachhinein.

Auch wenn FormAlchemy ursprünglich zur Nutzung in Kombination mit SQLAlchemy gedacht war, spricht nichts dagegen, auch dieses Framework „stand alone“ zu nutzen. Klassen mit Formularfeldern sind schnell angelegt, ebenso einfach ist die Einbindung von Validatoren. Formulare können wahlweise komplett oder feldweise generiert werden.

Fungiform ist trotz der niedrigen Versionsnummer recht komplett und bietet eine sehr durchdachte, praxisnahe Mischung aus eingebauten

und externen Validatoren. Weiterhin kennt das Framework einige spezielle Felder wie zum Beispiel für Komma-separierte Daten. Einziger Wermutstropfen ist der aktuell fehlende Maintainer und die dadurch bedingte vielleicht ungewisse Zukunft.

Flatland kann seine Stärken besonders bei komplexen und verschachtelten Formularen zeigen, da es hier deutlich mehr Möglichkeiten bietet, zumindest im Vergleich zu Fungiform, WTForms und FormAlchemy. Das Generieren von HTML schwankt zwischen einfach und eher umständlich. Ein echtes Manko ist die unvollständige Dokumentation.

Fazit

Welches Framework eingesetzt wird ist letztendlich Geschmackssache. WTForms, FormAlchemy und Fungiform sind in etwa auf Augenhöhe und gleichermaßen für den täglichen Einsatz geeignet. Deform ist etwas aufwändiger zu nutzen, da für das Anlegen der Formularklassen ein externes Modul genutzt wird. Bei komplexen, verschachtelten Formularen haben Deform und Flatland Vorteile, weil sie entsprechende Datenstrukturen direkt abbilden können. Beim Einsatz von Flatland ist aber aufgrund der lückenhaften Dokumentation etwas Pioniergeist seitens des Anwenders gefordert.

LINKS

[1] https://docs.pylonsproject.org/projects/deform/dev/

[2] https://docs.pylonsproject.org/projects/colander/dev/

[3] https://docs.pylonsproject.org/projects/peppercorn/dev/

[4] http://chameleon.repoze.org/

[5] https://docs.pylonsproject.org/projects/deform/dev/basics.html#validating-a-form-submission

[6] https://docs.pylonsproject.org/projects/deform/dev/common_needs.html#using-the-autocompleteinputwidget

[7] http://wtforms.simplecodes.com/

[8] http://wtforms.simplecodes.com/docs/0.6.2/

[9] http://wtforms.simplecodes.com/docs/0.6.2/ext.html

[10] http://code.google.com/p/formalchemy/

[11] http://www.sqlalchemy.org

[12] http://pypi.python.org/pypi/WebOb/1.1.1

[13] http://pypi.python.org/pypi/WebHelpers/1.3

[14] http://pypi.python.org/pypi/Tempita/0.5.1

[15] http://docs.formalchemy.org/fa.jquery/

[16] http://docs.formalchemy.org/pyramid_formalchemy/

[17] http://docs.formalchemy.org/

[18] http://www.pocoo.org/projects/fungiform/#fungiform

[19] http://www.pocoo.org/

[20] http://www.pocoo.org/projects/zine/#zine

[21] https://github.com/mitsuhiko/fungiform/blob/master/fungiform/forms.py

[22] http://de.wikipedia.org/wiki/ReCAPTCHA

[23] http://de.wikipedia.org/wiki/Cross-Site_Request_Forgery

[24] http://pypi.python.org/pypi

[25] http://discorporate.us/projects/flatland/

[26] https://bitbucket.org/jek/flatland/wiki/Todo

[27] http://discorporate.us/projects/flatland/docs/tip/

[28] http://rswilson.ch/flatland/

[29] http://discorporate.us/projects/flatland/docs/tip/signals.html

[30] http://discorporate.us/projects/flatland/docs/tip/templating/index.html

[31] http://www.formencode.org/en/latest/index.html

[32] https://www.djangoproject.com/

[33] https://docs.djangoproject.com/en/1.3/topics/forms/

[34] http://www.freiesmagazin.de/freiesMagazin-2011-02

Autoreninformation

Jochen Schnelle (Webseite) schreibt selber Web-basierte Python-Applikation und nutzt WTForms. Beim Blick über den Tellerrand entstand dieser Artikel zu den verschiedenen HTML-Formular-Frameworks.

Diesen Artikel kommentieren

Grafikadventures entwickeln mit SLUDGE

von Tobias Hansen

Adventure-Fans wird der Name SCUMM [1] sicherlich etwas sagen. SLUDGE [2] ist eine freie Alternative dazu und bringt selbst noch Entwicklerwerkzeuge mit. Mit diesem System kann man auf jeder Plattform Grafikadventures entwickeln und spielen. Der vorliegende Artikel soll einen kleinen Einblick in die ersten Schritte der Spielentwicklung mit SLUDGE geben.

Da die Geschichte von SLUDGE als Windows-Programm bereits mehr als zehn Jahre zurückreicht, sind schon einige relativ aufwändige Spiele entstanden. Weil SLUDGE mittlerweile auch auf anderen Plattformen erschienen ist, laufen diese Spiele nun natürlich auch unter Linux. Auf der SLUDGE-Homepage ist eine Liste der verfügbaren Spiele [3] zu finden. Der Wikiartikel zu SLUDGE auf ubuntuusers.de [4] enthält ähnliche Informationen auf deutsch.

Adventures entwickeln mit SLUDGE

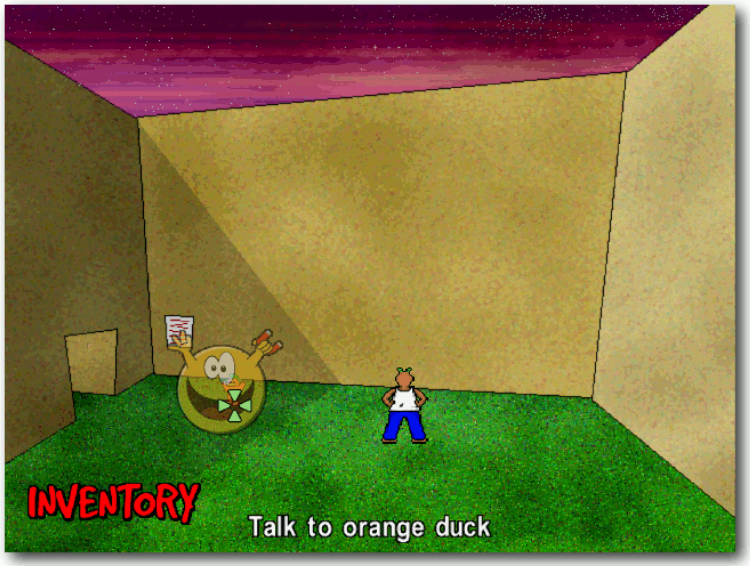
Der aufwändigste Teil bei der Entwicklung eines Adventures ist sicherlich der kreative – Story, Rätsel und Dialoge ausdenken, Hintergründe und animierte Objekte zeichnen, Musik, Sprache und Sounds aufnehmen. Dagegen ist das Zusammensetzen der Einzelteile ein Kinderspiel.

Um die mühsam erstellten Ressourcen in ein Spiel einzubinden und mögliche Interaktionen mit der Spielwelt einzuprogrammieren, wird bei SLUDGE eine Skriptsprache verwendet, die selbst auch SLUDGE (Scripting Language for Unhindered Development of a Gaming Environment) heißt. Außerdem besteht so die Möglichkeit, das Spielinterface völlig frei selbst zu gestalten.

Möchte man sich anfangs nicht mit dem Implementieren der Benutzerschnittstelle herumschlagen, sondern sofort den eigenen Charakter herumlaufen sehen, sollte man sich zu Beginn an dem Code eines funktionierenden Spiels orientieren. Hier bietet sich zum einen das im SLUDGE-Sourcecode-Archiv [5] (im Ordner `doc/ExampleProjects`) enthaltene „Verb Coin Example“ an.

Ein Beispielprojekt, das statt Verbmünzen-Interface ein klassisches SCUMM-Interface bietet, ist „The Interview“ (siehe Sourcecode-Bereich auf der SLUDGE-Homepage [6]).

Neben der Skriptsprache bietet SLUDGE fünf grafische Programme für recht spezialisierte Aufgaben. Drei davon dienen der Projektverwaltung, dem Erstellen von Übersetzungen und dem Umwandeln von Animationen in das von SLUDGE benötigte Format. Auf die anderen beiden



Verbmünzen-Interface beim „Verb Coin Example“.

SCUMM-Interface bei „The Interview“.

(Floor Maker und Z-Buffer Maker) wird weiter unten noch genauer eingegangen.

Ein ständiger Begleiter beim Entwickeln mit SLUDGE ist die Dokumentation [7]. Dort werden fast alle Fragen beantwortet.

Erste Schritte

Nun soll zur Demonstration der bei SLUDGE gängigen Konzepte beschrieben werden, wie mit dem Umwandeln des Verb Coin Example in ein eigenes Adventure begonnen werden kann.

Während der Einstiegspunkt für den Code in der Datei **init.slu** zu finden ist, beginnt der zum Spielinhalt gehörende Code in **outside/room.slu**. Wer sich etwas im Verb Coin Example umgeschaut hat, wird hier alles wiederfinden, was im ersten Raum passiert. Die Funktion **outsideRoom()** wird jedes Mal aufgerufen, wenn der Raum betreten wird. Da nun ein neues Spiel gestaltet werden soll, wird fast alles aus der Datei gelöscht, sodass nur Folgendes übrig bleibt:

```
1  sub outsideRoom () {
2      startMusic ('outside/music.xm', 0, 0);
3      addOverlay ('outside/room.tga', 0, 0);
4      setFloor ('outside/room.flo');
5      setScale (200, 150);
6      setZBuffer ('outside/room.zbu');
7
8      addCharacter (ego, 420, 400, makeEgoAnim ());
9      %~ say (ego, "Moin, moin! Scheint ja noch zu kompilieren der Code!");
10 }
```

Listing 1: room.slu

Die ersten fünf Zeilen laden Musik und Raum, dann wird der Spielcharakter angezeigt und zum Reden gebracht.

Damit das Projekt wieder kompiliert wird, müssen folgende Zeilen aus der Datei **iface/inventory.slu** gelöscht werden, da hier das Objekt **duck** referenziert wird, das soeben gelöscht wurde.

```
event duck {
    say (ego, "Hey, duck! Want a mushroom?");
    say (duck, "Er, no...");
    say (ego, "Ah well. I'll wander around with it for a little longer, ↷
    then.");
}
```

Nun kann das Projekt (**VerbCoin.slp**) mit dem Projekt-Manager geöffnet und **inside/room.slu** und **german.tra** aus dem Projekt entfernt werden. Jetzt sollte das Kompilieren wieder klappen.

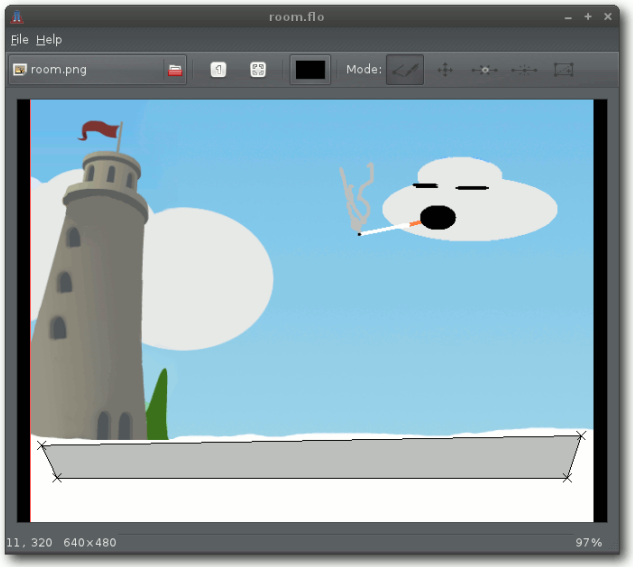
Startet man das neu kompilierte Spiel, gibt es keine Ente mehr in dem Raum. Auch kann man nicht mehr mit der Notiz oder dem Tunnel interagieren. Nun kann es an das Erstellen eines neuen Hintergrundes gehen.

Der erste eigene Raum

Jetzt beginnt die wirkliche Arbeit: Ein neuer Raum muss gestaltet werden. Das bevorzugte Dateiformat für Graphiken ist PNG, auch wenn das Verb Coin Example aus einer Zeit stammt, in der SLUDGE nur TGA verstand. Der Hintergrund wird in Zeile 3 in **outside/room.slu** ins Spiel eingebunden. Da sich die Endung im Dateinamen ändert, muss der Aufruf angepasst werden.

Dann muss festgelegt werden, wo der Protagonist herumlaufen darf. Dazu kann man einfach den vorhandenen Fußbodenplan **room.flo** im Floor Maker öffnen, das neue Hintergrundbild laden und die Ecken des Bodenbereichs zu den gewünschten Positionen ziehen.

Jetzt kann man im Spiel auf dem neuen Hintergrund herumlaufen. Allerdings ist der Protagonist viel größer als die Tür des Turmes, selbst wenn er direkt davor steht. Um einzustellen, wie groß Objekte an welcher Bildschirmposition sind, gibt es die Funktion **setScale()**, (Zeile 5 in



Floor Maker bei der Arbeit.

outside/room.slu). Die richtigen Werte herauszubekommen ist etwas knifflig, Hilfe gibt die SLUDGE-Dokumentation [8]. In diesem Beispiel wird folgendes verwendet:

```
setScale (370, 30);
```

Interaktion!

Diese Wolke sieht interessant aus. Und in Adventures möchte man sich interessante Dinge meist etwas genauer anschauen. **outside/room.slu** wird also wie im Listing2 erweitert.

In Zeile 8 wurde ein Bildschirmbereich für das Objekt **cloud** hinzugefügt. Die ersten vier Zahlen markieren die Koordinaten von zwei Ecken eines Rechtecks, das den Bereich definiert. Die anderen Zahlen geben an, wo der Protagonist hinlaufen soll, wenn mit der Wolke interagiert werden soll, und die letzte Angabe bestimmt, in welche Richtung er während der Interaktion guckt.

Die Koordinaten können ermittelt werden, indem das Hintergrundbild im Floor Maker geladen wird. Am unteren Rand des Floor Makers werden die Koordinaten des Mauszeigers angegeben.

Außerdem wurde das Objekt **cloud** erzeugt, mit je einer Aktion fürs Ansehen und Ansprechen.

Nichts wie raus hier

Zum Schluss soll noch ein Ausgang eingebaut werden. Der Protagonist soll auf Kommando in den Turm hineinlaufen, dann soll das Spiel beendet werden. Das funktioniert, ähnlich wie bei

```
1  sub outsideRoom () {
2      startMusic ('outside/music.xm', 0, 0);
3      addOverlay ('outside/room.png', 0, 0);
4      setFloor ('outside/room.flo');
5      setScale (370, 30);
6      setZBuffer ('outside/room.zbu');
7
8      addScreenRegion (cloud, 357, 65, 599, 161, 503, 380, NORTH);
9
10     addCharacter (ego, 420, 400, makeEgoAnim ());
11     say (ego, "Moin, moin! Scheint ja noch zu kompilieren, der Code!");
12 }
13
14 objectType cloud ("Merkwuerdig aussehende Wolke") {
15     event lookAt {
16         say (ego, "Komisch, die Wolke sieht so aus, als ob sie eine ↷
           Zigarette raucht.");
17     }
18     event talkTo {
19         say (ego, "Haaallooo Wooolkeee!");
20         pause(20);
21         turnCharacter (ego, SOUTH);
22         say (ego, "Na ja, so wie sie aussieht, scheint sie zu schlafen.")↷
           ;
23     }
24 }
```

Listing 2: room2.slu

Der Protagonist denkt ans Aufhören.

der Wolke, mit einer Kombination von ScreenRegion und Objekt. Dazu fügt man unter die bereits vorhandene **addScreenRegion**-Zeile in **sub outsideRoom()** die Zeile

```
addScreenRegion (intoTower, 75, 351, 115, 384, 127, 388, WEST);
```

ein. Ganz am Ende der Datei muss man dann noch das neue Objekt **intoTower** einfügen:

```
objectType intoTower ("Turmeingang") {
   event oneCursor = arrowEast;
   event onlyAction {
       forceCharacter (ego, 108, 382);
       forceCharacter (ego, 60, 382);
       pause(40);
       quitGame ();
      }
}
```

Die **forceCharacter**-Befehle bewirken, dass der Charakter aus seinem angestammten Fußbodenbereich heraus in die rechte Tür und dann etwas nach links läuft. Doch halt, nun scheint er vor dem Turm herumzufliegen. Hier wird ein Z-Buffer benötigt, damit der Charakter nur durch die Türöffnungen zu sehen ist, ansonsten aber hinter dem Turm verschwindet.

Es muss ein TGA-Bild mit schwarzem Hintergrund erstellt werden, in dem verschiedene Bereiche, hinter denen der Charakter verschwinden soll, einfarbig ausgemalt werden. Dieses Bild wird mit dem Z-Buffer Maker geöffnet und jeder Ebene wird eine y-Koordinate zugeordnet. Steht der Charakter mit den Füßen unterhalb dieser Koordinate, wird er vor der betreffenden Ebene gezeichnet. Sobald er die Linie überschreitet, wird er von dem entsprechenden Teil des Hintergrundbildes verdeckt.

Fazit

Nun sollte es dem geneigten Leser möglich sein, anhand der Dokumentation von SLUDGE und dem Beispielcode auch die Spielfigur zu ersetzen und vielleicht sogar ein eigenes kleines oder auch größeres Adventure zu verwirklichen.

Das hier im Artikel vorgestellte Beispiel kann zusammen mit allen wichtigen Projektdateien und Bildern als Archiv heruntergeladen werden.

LINKS

[1] http://de.wikipedia.org/wiki/Script_Creation_Utility_for_Maniac_Mansion
[2] http://opensludge.sourceforge.net
[3] http://opensludge.sourceforge.net/games.html
[4] http://wiki.ubuntuusers.de/OpenSLUDGE
[5] http://opensludge.sourceforge.net/download.html
[6] http://opensludge.sourceforge.net/resources.html#sourcecode
[7] http://opensludge.svn.sourceforge.net/viewvc/opensludge/doc/SLUDGE_Help.html
[8] http://opensludge.svn.sourceforge.net/viewvc/opensludge/doc/SLUDGEDevKitHelp/Scaling_Characters.html

Autoreninformation

Tobias Hansen spielte 2004 „Out Of Order“ und nennt das SLUDGE-Spiel seitdem als eines seiner Lieblings-Fanadventures. Anfang 2010 stieß er zufällig wieder auf das inzwischen quelloffene SLUDGE und beschloss, es nach Linux zu portieren.

Diesen Artikel kommentieren

PHP-Programmierung – Teil 3: Arrays, Sessions, Sicherheit

von Patrick Eigensatz

Im vorangegangenen Artikel (siehe freiesMagazin 11/2011 [1]) ging es um Variablen, Kontrollstrukturen und Funktionen in PHP. Nun wird etwas mehr auf Datenfelder eingegangen. Weiterhin wird noch gezeigt, wie man sich ein sicheres Log-in programmiert, in welchem man eingeloggt bleibt. Am Ende des Artikels wird außerdem auf Sicherheitsprobleme mit PHP hingewiesen.

Auflösung der Escape-Aufgabe

Die Escapeaufgabe aus dem letzten Artikel war:

```
<?php
$var1 = '\'\'\'';
$var2 = "\"\"" . '"';
echo "$var1$var2$var1\"";
?>
```

Wenn man die maskierten Zeichen auflöst und die Strings nicht mit Anführungszeichen umgibt, ergeben sich folgende Werte für **$var1** und **$var2**:

```
$var1 = ''' // drei einfache Anführungszeichen
$var2 = """ // drei doppelte Anführungszeichen
```

Zusammen ergibt das (ohne Leerzeichen):

```
' ' ' " " " ' ' '
```

Hinzu kommt das maskierte doppelte Anführungszeichen, das beim **echo**-Befehl angefügt wurde. Die Lösung ist also:

```
' ' ' " " " ' ' ' "
```

Arrays

Arrays wurden bereits in einem der vorherigen Teile vorgestellt, sollen in diesem Teil aber etwas näher behandelt werden. Bei Arrays handelt es sich um Variablen, die mehrere Werte speichern können. In den vorhergehenden Teilen wurden Arrays als „Hängemappenregister“ vorgestellt. Es wird unterschieden zwischen Listen-Arrays und assoziativen Arrays.

Listen-Arrays

Man stelle sich vor, man will 20 Zufallszahlen zwischen 1 und 100 generieren. Die Funktion, um eine Zufallszahl zu generieren, heißt **rand**. Man wendet sie zum Beispiel so an:

```
$zahl1 = rand(1, 100);
$zahl2 = rand(1, 100);
...
$zahl19 = rand(1, 100);
$zahl20 = rand(1, 100);
```

Die Nachteile dieses Codes sind die Unübersichtlichkeit und der Platzverbrauch. Besser geht es mittels einer **for**-Schleife und einem Array:

```
$zufallszahl = array();
for($nummer = 1; $nummer <= 20; $nummer++) {
    $zahl = rand(1, 100);
    array_push($zufallszahl, $zahl);
}
```

Dieser Code bewirkt fast dasselbe. Anstelle der Definition von 20 Variablen wird eine „Sammelvariable“ definiert. Dies geschieht mit

```
$zufallszahl = array();
```

Dann wird die Schleife solange ausgeführt, wie **$nummer** kleiner oder gleich 20 ist. **$nummer** wird bei jedem Durchgang um eins erhöht. Während jedes Durchgangs ist **$zahl** eine neue Zufallszahl. Diese wird mithilfe von **array_push** in ein Array eingefügt. Ist die Schleife fertig, kann man, ähnlich wie bei **$_POST** oder **$_GET**, über die eckigen Klammern auf die Werte zugreifen. Der einzige Unterschied besteht darin, dass hier Zahlen zwischen den Klammern stehen und keine Zeichenketten.

```
echo $zufallszahl[0];
```

Gibt z. B. die erste Zufallszahl aus.

Assoziative Arrays

In diesem Abschnitt werden die zuvor angetroffenen assoziativen Arrays vorgestellt. Auch in PHP gibt es die Möglichkeit, eine Art „Übersetzung“ zu speichern. Angenommen, man hat ein assoziatives Array **$benutzer**. In **$benutzer** werden verschiedene Werte abgespeichert, auf die man mithilfe der eckigen Klammern und einem sogenannten Schlüssel dazwischen zugreifen kann: **$benutzer[SCHLÜSSEL]**. **$benutzer** kann man als Datensatz einer Datenbank ansehen. Ihm können beliebig viele Informationen zugeordnet

werden, immer über die eckigen Klammern und den Schlüssel, z. B:

```
$benutzer['betriebssystem']
$benutzer['geburtstag']
$benutzer['lieblingsessen']
```

Außerdem kann man assoziative Arrays für alles verwenden, was irgendwie „übersetzt“ werden muss:

```
$hauptstadt["Schweiz"] = "Bern";
$franz["das Dorf"] = "le village";
```

Zur Übersetzung von Wörtern zwischen Sprachen, hier ein kurzes Beispiel:

```
<?php
// Definiere Wortschatz:
$englisch = array("Buch" => "book", "Kabel" => "↷
cable", "Kuh" => "cow");
// Lese das übergebene Wort ein:
$de_wort = $_GET['wort'];

// Gib die Übersetzung aus:
echo $englisch[$de_wort];
?>
```

Man ruft das Skript dann mittels **uebersetzung.php?wort=Buch** auf und erhält:

```
book
```

In der dritten Zeile wird das Array definiert. Hier werden der Funktion aber Parameter übergeben. Diese werden speziell getrennt. An erster Stelle kommt der Schlüssel, über den auf den nach dem Pfeil folgenden Wert zugegriffen werden kann. Das wiederholt sich für jeden Eintrag im Array. Die einzelnen Einträge werden durch Kommas abgetrennt. Hier wieder ein Beispiel:

```
"schluessel" => "wert", "↷
schluessel2" => "wert2", "↷
schluessel3" => "wert3"
```

Sicherlich kann man auch auf die Idee kommen, dem Skript ein Wort zu übergeben, das gar nicht definiert ist. So wird mit dem Aufruf **uebersetzung.php?wort=Tisch** gar nichts ausgegeben, da die Variable **$englisch['Tisch']** nicht existiert. Mit **isset()** kann man aber überprüfen, ob eine Variable vorhanden ist oder nicht:

```
<?php
// Definiere Wortschatz:
$englisch = array("Buch" => "book", "Kabel" => "↷
cable", "Kuh" => "cow");
// Lese das übergebene Wort ein:
$de_wort = $_GET['wort'];

// Überprüfe, ob das Wort im Array vorhanden ist
if(isset($englisch[$de_wort])){
    // Wenn ja, gib die Übersetzung aus:
    echo $englisch[$de_wort];
} else {
    // Sonst, gib aus:
    echo 'Ich kenne das Wort nicht!';
}
?>
```

Wer jetzt verstanden hat, was assoziative Arrays sind, wird sicher erkennen, dass **$_GET** und **$_POST** nichts anderes als assoziative Arrays sind. Nicht wesentlich anders sieht es im nächsten Abschnitt aus.

PHP-Sessions

PHP-Sessions sind Sitzungen, die meist nach einem Log-in mit dem Server aufgebaut werden, damit dieser einen später auf anderen Seiten wiedererkennen kann. Wenn man sich einloggt, möchte man nicht, dass man sich nach jedem Klick auf einen Link wieder einloggen muss. Dazu wird eine 32 Byte lange, zufällige Kombination aus Hexadezimalzeichen erstellt. Der Computer des Benutzers, der sich gerade eingeloggt hat, speichert diese in einer kleinen Datei (genauer gesagt in einem Cookie). Wenn der Anwender das nächste Mal einem Link folgt, wird die kleine Datei mit dem Aufruf der Seite mitgesendet. Dadurch kann der Server diesen Benutzer eindeutig identifizieren. Damit der Browser die Cookies akzeptiert und abspeichert, müssen sie in den Einstellungen aktiviert sein.

In einer Session können beliebig viele Informationen in einem assoziativen Array abgespeichert werden. Das Array dazu lautet **$_SESSION**. Wer mehr über Sessions erfahren möchte, findet Informationen im entsprechenden Wikipedia-Eintrag [2]. Informationen über die Sicherheit findet man im PHP-Manual [3].

Damit PHP übermittelte Session-Cookies einliest oder dem Client eine neue ID zuteilt, muss jede Seite, bei der man auf die Session-Daten Zugriff haben möchte, mit einem

```
session_start();
```

gestartet werden. Da dies eine Funktion ist, die auf den Protokollheader zugreift, muss sie vor jeder Ausgabe ausgeführt werden. Ich persönlich schreibe die **session_start()** immer gleich

nach dem **<?php**. Das bereitet am wenigsten Probleme. Hier ein Beispiel:

php3_session.tar.gz

In der Datei **erfassen.html** befindet sich das Formular, über das der Name mitgeteilt wird. Dort wird zuerst überprüft, ob bereits ein Name gesetzt ist, dann wird dieser mit dem angegebenen Namen überschrieben. Folgt man zu **seite2.php** wird dort der Name angezeigt, sofern er gesetzt ist. In **logout.php** wird die Session zuerst eingelesen, damit sie eine Codezeile später zerstört werden kann. Die Session muss immer zuerst gestartet werden, damit sie beendet werden kann. Mit dem Zerstören der Session gehen alle in **$_SESSION** gespeicherten Daten verloren! Gerne kann man in diesem Beispiel auch zwischen den Seiten herumhüpfen, also einfach mal auf **logout.php** gehen, obwohl man nicht eingeloggt ist. Das Skript ist gegen solche Spielereien geschützt, indem es mit **isset()** die gesetzten Variablen überprüft.

Sicherheit in PHP-Skripten

Wie man sieht, ist der Gebrauch der Funktion **isset()** sehr wichtig. Sie ist fast immer nötig, damit keine unschöne Fehlermeldung in der Webseite angezeigt wird, wenn eine Variable nicht gesetzt ist. Ein häufiger Fehler ist die unterschiedliche Schreibweise von zwei Variablen (z. B. **$pw** und **$passwort**). Und dabei stößt man langsam auf ein Problem der Programmiersprache PHP. Die Existenz von Variablen ist nicht sicher. In einem C-Programm kann man sicher sein, dass Variablen existieren – auch wenn sie keinen brauchbaren Wert haben. Sie können normalerweise nicht einfach gelöscht werden.

Der Zugriff auf undefinierte Variablen kann, je nach deren Wichtigkeit, ein PHP-Skript zum Scheitern bringen. Deshalb sollte man bei allen Variablen, die der Nutzer irgendwie beeinflussen kann (nicht eingeloggt sein oder die GET-Variablen, bzw. die URL-Zeile modifizieren), deren Existenz überprüfen.

Wichtig ist außerdem, dass man Eingaben, die irgendwo per **echo** ausgegeben werden, filtert. Gibt man als Name in einem Formular HTML-Code ein, wird dieser per **echo** auf die Seite geschrieben, und der Browser interpretiert das HTML. Der Benutzer kann also HTML einfügen und interpretieren lassen. Es kann weitaus mehr passieren, wenn JavaScript-Code eingefügt wird. Dieser könnte zum Beispiel auf eine mit Schadcode behaftete Webseite weiterleiten. Am besten ist es, den HTML-Tag-Anfang (<) zu filtern, denn dann werden HTML-Tags nicht mehr interpretiert. Um eine Zeichenkette zu ersetzen, kann man **str_replace** benutzen.

```
<?php
echo "<!DOCTYPE html>
<html><body><p>";

$test = $_GET['xsstest'];
$test = str_replace("<", "&lt;", $test);
$test = str_replace(">", "&gt;", $test);

echo "Die Eingabe war $test";
echo "</p></body></html>";
?>
```

Bei **<** und **>** handelt es sich um HTML-Zeichen für „kleiner gleich“ und „größer gleich“.

Natürlich bietet PHP eine einfachere Funktion:

```
<?php
$test = htmlspecialchars($_GET['xsstest']);
echo "Die Eingabe war " . $test;
?>
```

Fazit

Nach diesem Abstecher in die PHP-Sicherheit sollte man sich merken, dass man in PHP alles filtern und überprüfen sollte. PHP ist keine Programmiersprache für Programmierer, die ein sicheres Programm möglichst „rasch“ schreiben wollen. Aber: Alle Programmierer machen irgendwo Fehler.

LINKS

[1] http://www.freiesmagazin.de/freiesMagazin-2011-11

[2] http://de.wikipedia.org/wiki/Session-ID

[3] http://www.php.net/manual/de/session.security.php

Autoreninformation

Patrick Eigensatz (Webseite) befasst sich seit einigen Jahren mit der Entwicklung von Webanwendungen und hat dadurch viele Erfahrungen im Bereich PHP gesammelt.

Diesen Artikel kommentieren

Perl-Tutorium – Teil 4: Referenzen auf Arrays und Hashes

von Herbert Breunung

Viele wichtige Perlvokabeln wurden bereits in den Teilen 1 bis 3 beschrieben und angewendet. Es ist also Zeit, einen Gang runter zu schalten, um Details zu betrachten, die bisher überflogen wurden. Zum Beispiel werden Arrays und Hashes gründlicher verglichen und wesentliche Ausgabehilfen vorgestellt. Ziel dieses Textes ist es aber zu zeigen, wie sie sich wie Legosteine kombinieren lassen, um die Datentürme noch höher und breiter zu bauen. Das Demoprogramm wird dafür kurzfristig in den Hintergrund treten.

Nachträge

Eigentlich stünde der letzten Folge der Titel „String- und Listenmanipulation sowie Hashes“ wesentlich besser, denn weder Schleifen noch Subroutinen wurden groß erörtert. Dafür aber beide Möglichkeiten der bedingten Ausführung (**if** und **when**). **for**-Schleifen wurden erläutert, aber es gibt noch zwei weitere Arten: **while** und **until**. Rein logisch verhalten sie sich wie **if** und **unless**. Bei einer einfachen **if**-Anweisung (ohne **elsif** oder **else**) wird der Block ausgeführt, wenn die Bedingung einen positiven Wert (nicht 0, leer oder **undef**) zum Ergebnis hat. Ebenso verhält sich **while**. Nur prüft es danach wieder die Bedingung, sooft bis sie einmal negativ ausfällt. Erst danach wird der Block nicht mehr ausgeführt. Damit lassen sich also Programme schreiben, die niemals ein Ende finden. Manchmal ist das allerdings erwünscht. Dann wird oft so etwas geschrieben:

```
while (1) {
    say "Ich dreh hier schon seit ↷
    Stunden ...";
}
```

until funktioniert genauso, verneint (wie mit **not**) nur das Ergebnis der Bedingung, so wie **unless** es auch tut. Es führt solange den Block aus, bis die Prüfung positiv ausfällt. Lustigerweise kennen fast alle Sprachen (bis auf Python) **until**, aber neben Perl nur wenige (wie Ruby) auch **unless**.

Und die im Einleitungsteil gepriesene derzeitige Umgestaltung der Infrastruktur schritt auch seit freiesMagazin 07/2011 [1] gut voran. Die damals aufgezählten CPAN-Dienste, wie etwa Bewertungen oder die Testmatrix, die über mehrere Seiten verstreut sind, können jetzt über die MetaCPAN-Seite [2] auf der Hauptseite jedes Moduls in zwei Spalten überblickt werden. Welche Module das angezeigte benötigen, wird aufgelistet. Sogar die Entwicklung der Quellen ist dort graphisch dargestellt und es lässt sich einfach in den Versionsunterschieden blättern. Die Suche wurde bequemer, da wie von Google gewohnt, während des Tippens Vorschläge unterbreitet werden. Die Modul-URL wurden ebenfalls kürzer und prägnanter. Auch das Angebot von PrePAN [3] wird gerade vom MetaCPAN aus verknüpft. Auf dieser kürzlich erst gegründeten Seite können Entwickler und Interessierte über vorhandene und geplante (daher der Name „Pre“) Module diskutieren. Wem die Gelegenheit fehlt, die ungezählten Perlblogs täglich im Blickfeld zu behalten, der informiere sich in etwa fünf bis zehn Minuten mit der wöchentlichen Rundmail von Perl Weekly [4], die Gabor Szabo nun versendet. Sie ist lesbarer und informativer als die Übersichten der letzten wichtigereren Blog-Beiträge, die Andy Lester auf Perlbuzz [5] zusammenträgt.

Der letzte Neuzugang ist das Perl Tutorial Hub [6], ein Zentralregister für alle Lehrmaterialien zu Perl, welche dort sortiert und bewertet werden. Auch dieses Tutorium wurde dort bereits erfasst, aber noch nicht bewertet.

Variablen

Das Thema Variablen ist in Perl so simpel wie möglich gehalten. Jede skalare Variable kann ohne Voranmeldung alles aufnehmen, egal ob Zahlen, Texte, Entscheidungen, Suchmuster oder Programmteile. Es ist auch nicht wichtig, wie groß die Daten sind oder ab wann sie nicht mehr gebraucht werden. Der Interpreter regelt die technischen Details.

Manchen gefällt nur nicht, dass Variablen mit **$**, **@** oder **%** anfangen müssen. Aber anderen Menschen fällt es so leichter, Variablen wiederzuerkennen. Ebenso kann Perl (wie im letzten Teil gezeigt) damit die Variablen in jeder Lage

(wie etwa innerhalb Anführungszeichen) ohne Zusätze und Verrenkungen wiedererkennen. Die Systemadministratoren waren das eh von ihrer Shell gewohnt. Außerdem erlaubt es eine Variable **$read_file** zu haben haben, die keine Probleme mit der Funktion **read_file()** des Moduls **File::Slurp** hat. Wobei **$read_file** kein wirklich guter Variablenname ist. Gute Namen beschreiben den Inhalt so genau wie möglich, wie etwa **$maerchentext** oder **$laenge_in_m**. Das ist eine wesentliche Zutat für Programme, die auch noch nach Jahren verständlich bleiben.

$read_file wäre schon deshalb kein guter Variablenname, weil er eine Tätigkeit beschreibt. In Programmen sind es aber eher die Funktionen (in Perl Subroutinen genannt), welche für Handlungen stehen und damit die Rolle von Verben haben. Die Variablen zeigen womit und wie gehandelt wird. Sie sind die Substantive und Adjektive. Nicht jeder sieht das so, aber der Linguist Larry Wall schon. Für ihn symbolisieren **$** und **@** die Einzahl- und Mehrzahl-Endungen der Substantive. Deshalb bekommt man mit **@notizen** alle Elemente als Liste. Und deshalb schreibt man auch **$notizen[0]**, um das erste Element zu erhalten, was nichts mit einer vielleicht existierenden Skalarvariable **$notizen** zu tun hat. Bei Notiz eins und zwei heißt es wieder **@notizen[0,1]**. **%** ist nur eine besondere Mehrzahl, bei der die Glieder in Paaren auftreten. Dieses Konzept war bis Perl 4 sinnvoll. Ab Perl 5 bereitet es Einsteigern darüber hinaus einige Schwierigkeiten, die erst ab vorigem Jahr mit 5.14 langsam abnehmen. Die Ursache dafür sind Referenzen, die neben Modulen wohl wichtigste Neuerung, die Perl 5 vor 17 Jahren brachte. Seit es sie gibt, verschwand die Garantie, das **$** für einen Einzelwert steht.

Einfache Referenzen

Kommt ein Postbote an die Haustür und liest dort: „Achtung! Mieter verzogen – die neue Anschrift ist Nymphenweg 10, 1234 Lärchenhain“, weiß er, wohin das Paket gehen soll. Mit den Referenzen ist das ähnlich. Das sind Adressen die besagen: „Was du suchst, ist dort drüben in der Speicherzelle für Skalare Nummer soundso.“ Das heißt dann ausgeschrieben **SCALAR(0x8c82eb0)**. Im normalen Gebrauch sind Referenzen aber recht einfach.

```
my $original = "Alles paletti Egon.";
my $kopie = \$original;
```

Der Backslash erzeugt die Referenz auf das was ihm folgt. In **$kopie** ist damit eine Referenz auf **$original**. **say $kopie;** bestätigt das mit der Ausgabe von **SCALAR(0x........)**. Um die Referenz aufzulösen, also der Adresse nachzugehen muss ein weiteres **$** vor **$kopie** gesetzt werden.

```
say $$kopie; # Alles paletti ...
```

Bereits das erste Dollarzeichen besagte: „Schau nach, ob du im Zentralregister (Symboltabelle) unter dem Namen **kopie** eine Skalarvariable findest und gib mir den Inhalt.“ Das Ergebnis ist die erwähnte Adresse ohne schönes, selbstgewähltes Pseudonym. Das zweite **$** schickt den Boten erneut mit der erhaltenen Adresse und dem gleichen Befehl los, worauf der zu berichten weiß, dass alles paletti ist. Um herauszubekommen welcher Art die Referenz ist, gibt es den Befehl **ref**:

```
say ref $kopie;
```

Das gibt **SCALAR** aus, den meistens interessanten Teil der Adresse. Wie einfach zu erraten ist, antwortet der Befehl in anderen Fällen **ARRAY** oder **HASH**.

```
say ref \@notizen';  # ARRAY
say ref \%kommando'; # HASH
```

ref $$kopie liefert gar nichts (einen leeren String, kein **undef**). **$$kopie**, gleichbedeutend mit **$original**, enthält nämlich keine Referenz, sondern einen einfachen Wert. In manchen Programmen werden nur die **$kopie** verwendet, aber nicht das **$original**. Da mag sich mancher wundern, warum man nicht gleich

```
my $kopie = \"Alles paletti Egon.";
```

schreibt. Die Antwort ist, weil es ein Sonderfall ist. Das erzeugt eine Referenz auf einen konstanten Wert und nicht auf eine Stelle im Speicher. Wer dennoch versuch **$$kopie** zu ändern, ohne ein **$original** anzulegen, wird sofort mit einem Programmabbruch belohnt, gekrönt von der Meldung: „Modification of a read-only value attempted at . . . “

Referenzen auf Arrays

Interessanter wird es, wenn die Referenz auf einen Array zeigt. Dafür stehen zwei Wege zur Auswahl:

```
my @primzahlen = (2,3,5,7,11,13;17)↷
;
my $aref = \@primzahlen;
my $aref = [2,3,5,7,11,13,17];
say ref $aref;  # sagt: "ARRAY"
say "@$aref"; # "2 3 5 7 11 13 17"
```

Um diese Referenz aufzulösen, braucht es wieder das passende Sigel. Gut, dass mit **ref** nachgefragt wurde, welcher Art die die Referenz war. Aber kann man so auch ganz normal auf das Array zugreifen? Eindeutig: Ja! Sogar ohne dass man das Sigel noch dazu verändern müsste, wenn man ein oder mehrere Elemente des Array anspricht. **@$pref[....]** wäre in jedem Fall korrekt. Der übliche Dienstweg ist aber ein anderer, nämlich der Pfeil-Operator:

```
say @$pref[1];    # zweites Element
                  # ist 3
say $pref->[1];   # dito
$pref->[1] = "darf ich?";
```

Und dieses Mal läuft Perl wie am Schnürchen, da bei Arrays und Hashes die referenzierten Inhalte nicht als Konstanten angelegt werden. Die dafür zu verwendenden Klammern lassen sich einfach merken. Eckige Klammern erzeugen eine Referenz auf ein Array der umschlossenen Werte. Denn um Teile (Slices) eines Arrays zu bekommen, nimmt man auch die eckigen. Genauso einheitlich ist die Verwendung der geschweiften Klammern bei Hashes.

Wie bereits angedeutet, kann man sich ab Perl 5.14 das Dereferenzieren in einigen Situationen sparen. Auf deutsch: Wenn Befehle, die eh nur bei Arrays sinnvoll sind, einen Skalarwert bekommen, schauen sie nach, ob es eine Referenz auf einen Array ist. Im Beispiel:

```
# nächste Primzahlen zufügen:
push @$pref, 19, 23;
# mit use v5.14; :
push $pref, 19, 23;
```

Neben **push** kann das auch noch **pop**, **shift**, **unshift**, **splice**, **keys**, **values** und **each** Verwendung finden. Die letzten drei lösen auch Hashref auf.

Referenzen auf Hashes

Fast alles über Arrayreferenzen Gesagte ließe sich noch einmal für Hashreferenzen wiederholen.

```
my %autoren = (
    mjd => 'hop', dc => 'pbp', c => 'mp'
);
my $href = \%autoren;
my $href = {
    mjd => 'hop', dc => 'pbp', c => 'mp'
};
say $href->{'mjd'} # sagt: "hop"
say ref $href;  # sagt: "HASH"
```

Hashes sind wie Arrays eine Sammlung von Skalarwerten. Nur hat in einem Hash jeder Wert einen vom Programmierer gewählten Namen, der erst einmal als Text verstanden wird und keine Nummer wie im Array. Diese Namen (Schlüssel) besitzen auch keine feste Reihenfolge. Deshalb wäre es fruchtlos, **splice** auf einen Hash anzusetzen. Um ein Paar zu löschen, bedarf es nur einem

```
delete @g{'mjd'};
```

Die kryptisch anmutenden Inhalte des Hashes sind Literaturempfehlungen. Mark Jason Dominus schrieb „High Order Perl“ um zu zeigen, dass Perl sehr gut als funktionale Programmiersprache taugt. Damian „larrys evil henchman“ Conway gab mit „Perl Best Practice“ eine vielbeachtete Stilvorgabe und chromatic schrieb vor zwei Jahren „Modern Perl“, um zu unterstreichen, was heute den meisten anderen Büchern fehlt.

Arrays ausgeben

Man sollte hier aber noch einmal einen Blick zurück werfen:

```
say "@$aref";
#sagt "2 3 5 7 11 13 17"
```

Auch ein **say "@notizen";** setzt klärend Leerzeichen zwischen die Inhalte der Elemente. Wer lieber Bindestriche hätte, erreicht das mit einem vorherigen **local $" = '-';** Es gibt auch noch die magische Variable **$**, die im Normalfall **""**, also keinen Text, beinhaltet. Würde man sie mit einem Leerzeichen füttern,

```
local $, = ' ';
say "Ihre Nachricht:", $notiz;
```

würde dieses **' '** zwischen **Ihre Nachricht:** und dem Inhalt der **$notiz** eingefügt werden.

Manche halten jedoch die Verwendung von „magischen" Spezialvariablen für ein Werk der Finsternis. Natürlich kann sie unerwünschte Nebeneffekte haben, wie bereits im Teil 2 (Abschnitt „Lösung der Aufgabe") erläutert. Verändert man aber (wie in diesem Beispiel) mit **local** nur eine lokale Kopie der Variable, deren Lebensdauer bis zum Ende des aktuellen Blocks reicht, werden die Nebeneffekte vermieden. Wem es dennoch verboten ist oder wer es für böse hält, kann den Effekt von **local $" = '-'** auf einem anderen Weg erzielen:

```
say "Ihre Nachrichten: ",
    join( '-', @primzahlen );
# sagt: "2-3-5-7-11-13-17"
```

join ist das Gegenteil des im letzten Teil vorgestellten **split**. **split** bekam als ersten Parameter ein Textstück oder Suchmuster und teilte anhand dessen den Inhalt des zweiten Parameters in Teile. Das Ergebnis war ein Array. **join** kann die Teile wieder verbinden und zwischen sie wieder die entnommenen Trennzeichen fügen.

Hashes ausgeben

Mit

```
say "%$href";  # sagt: "%$href"
```

erhält man den ungewollten, unschön formatierten Ausdruck von Hashdaten. Damit ist Perl 5 auch etwas überfordert. Um den Inhalt von Hashes übersichtlich aufzulisten, gibt es das Modul **Data::Dumper**:

```
use Data::Dumper;
print Dumper( $href );     # oder:
print Dumper( \%autoren );
```

Dies gibt aus:

```
$VAR1 = {
  'c' => 'mp',
  'dc' => 'pbp',
  'mjd' => 'hop'
};
```

Wem das zu viel Platz wegnimmt und wer nicht für jede öffnende oder schließende Klammer und jedes Wertepaar eine Zeile nutzen will, der nehme **Data::Dump**. Das spart auch das **print**, **\n** oder **say**, verlangt aber vom Benutzer, das Modul erst einmal zu installieren. **Data::Dumper** ist im Unterschied dazu immerhin ein Kernmodul und bei jedem Perl von Anfang an dabei.

```
use Data::Dump qw(dump);
dump(\%autoren);
# sagt: { c => "mp", dc => "pbp", ↷
mjd => "hop" }
```

Dieses Format ist das von Perl. Und (wie auch **Data::Dumper**) besonders praktisch, wenn man überprüfen will, was das Programm gerade macht und welche Daten wirklich in der Variable stecken, oder ob sich ein Fehler eingeschlichen hat. Wenn man Hashes oder noch Komplizierteres in eine Datei ablegen will, um sie beim nächsten Programmstart wiederherzustellen, gibt es andere Module für diese Arbeit. Die bekanntesten sind JSON und YAML. Das sind beides international verbreitete Standards, für die viele Sprachen auch Bibliotheken haben, so auch Perl. Für das Notizprogramm wäre YAML geeigneter. Es ist mächtiger und kann sogar Strukturen erkennen, in denen zwei Referenzen auf die gleiche Speicherstelle zeigen.

Arrays von Arrays (AoA)

Bisher konnten die Referenzen nicht ihren vollen Nutzen zeigen. Mit einem einzelnen Zeiger auf einen Array beeindruckt man keinen praktisch denkenden Menschen. Was wäre aber, wenn jeder Wert eines Arrays auf einen anderen Array zeigt. Dann hätte man nicht nur eine Zeile von Werten sondern eine Fläche. Es ist wie das Koordinatensystem im Mathe-Unterricht. Zuerst kommt die X-Koordinate (Spalte, Nummer im Oberarray) und dann Y (Zeile, Nummer im Unterarray). Für ein kleines 3x3-Feld wäre das:

```
my $magisches_quadrat = [
  [8,1,6],
  [3,5,7],
  [4,9,2],
];
```

Der zweite Wert des ersten Arrays ist 1 (der Koordinatenursprung liegt links oben). Um ihn zu erreichen, schreibt man

```
say $magisches_quadrat->[1]->[0];
# sagt: 1
say $magisches_quadrat->[1][0];
# sagt: 1
```

Den zweiten Pfeil kann man weglassen, da Perl auch so weiß, was gemeint ist. Nur den ersten Pfeil muss man dazu schreiben, um Zweideutigkeiten zu vermeiden. Folgendes ist ein völlig anderer Ausdruck:

```
say $magisches_quadrat[0][0];
```

Das erste Beispiel war ein Skalar, der auf ein Array zeigt. Im zweiten Beispiel bekommt man ein Element eines Arrays. Zufällig ist es eine Array-Referenz, von der wiederum das erste Element verlangt wird. Das ist eine Struktur, die so definiert wurde:

```
my @magisches_quadrat = (
  [8,1,6],
  [3,5,7],
  [4,9,2],
);
```

Jetzt lichtet sich auch der Nebel um die Aussage, dass Referenzen die Bedeutung der Sigel stören. **$magisches_quadrat[0]** sieht nach einem Skalar aus, ist aber in Wirklichkeit ein Array. Oder zumindest eine Referenz auf einen Array.

Richtig unelegant wird es, möchte man das Array, auf den **$magisches_quadrat[0]** zeigt, in eine andere Array-Variable kopieren:

```
my @zeile = @{ $magisches_quadrat[0] };
```

Hier reicht es nicht, den gewollten Kontext zu erzwingen, indem man nur ein **@** davor setzt wie bei **@$aref**. Die geschweiften Klammern müssen klarstellen, dass nicht **$magisches_quadrat**, sondern **$magisches_quadrat[0]** in den Array-Kontext gesetzt werden muss. Jetzt kann das **=** die Brücke spannen und Klone der Schäfchen auf der Weide rechts wandern brav über den Bach in **@zeile** hinein.

```
my $zeile = $magisches_quadrat[0];
```

Das wäre zwar einfacher, aber in dem Fall erhält man nur die Adresse einer Zeile (Unterarray) in **@zeile**. Wird ein Schubfach von **$zeile->[0]** aus entleert, gibt es auch nichts mehr von **$magisches_quadrat[0][0]** aus gesehen.

Hashes verstöpseln

Hashes in Hashes, Hashes in Arrays, gerichtete Graphen, die Wege über Königsberger Brücken, fast jede erdenkliche Datenstruktur kann gebaut werden, wenn man nach den gleichen Regeln auch Referenzen auf Hashes verwendet. Als abschließendes Beispiel sei eine Struktur erzeugt, die Spielstände eines Schachspiels speichern kann. Dazu braucht es einen Hash mit den Schlüsseln **A** bis **H** (die Spalten). Jeder Wert ist eine Referenz auf ein Array der Länge 8 (**0..7**, die Zeilen). Natürlich könnte man auch ein Array mit Hash-Referenzen aufziehen, aber so lässt sich später schreiben:

```
say $feld{'A'}[3];
# sagt: "Dame"
```

Das entspricht der gewohnten Sprache der Schachspieler und erzeugt weniger Verwirrung. Hochtrabend wird dieses Prinzip auch „Domain-Driven Design“ (DDD) genannt und sagt eigentlich nicht mehr als: „Schreibst du für jemand ein Programm, dann verwende seine Logik und seine Fachvokabeln.“

Perl hat die wunderbare Eigenschaft, eine solche Struktur nicht vorbereiten zu müssen. Nach einem simplen **my %feld;** könnte man die Abfrage **$feld'A'[3];** machen und bekommt nur ein korrektes **undef** als Ergebnis. Man ist hier halt weniger streng als anderswo. Während dieser Aktion geschah aber heimlich, still und leise noch etwas, was Außenseiter nicht erwarten. Um im Unterarray nachsehen zu können, wurde im Hash der Schlüssel **A** angelegt und in seinem Wert die Referenz auf einen ebenfalls neu erschaffenes, leeres Array abgelegt. In kaum einer anderen Sprache gibt es diese „Autovivifikation“. Sie erlaubt unvorbereitete Zuweisungen wie **$feld'D'[55] = 'Läufer';**, sagt aber leider auch niemals nein, selbst wenn eine Zeile 55 ausgesprochen unerwünscht ist.

Noch ein letzter Hinweis: Ist es dennoch gewünscht, die beschriebene Datenstruktur aufzubauen, sind **for** und ähnliche Befehle sehr nützlich. Anstatt alles auszuschreiben

```
my %feld = (
    A => [0,1,2,3,4,5,6,7],
    B => [0,1,2,3,4,5,6,7],
    ...
```

lässt sich das auch kürzer schreiben. Auch in der Lage kann **0..7** das **0,1,2,3,4,5,6,7** ersetzen. Darüber hinaus funktioniert der Bereichsoperator **..** auch mit Buchstaben. **'A'..'H'** erzeugt ein Array mit den gebrauchten Spaltennamen, der mit **for** abgearbeitet werden kann:

```
my %feld;
$feld{ $_ } = [0..7] for 'A' ..'H';
```

Nur bitte das **my %feld;** nicht vergessen, **use strict;** ist aktiviert (siehe Teil 1). Auch die Schleife ist notwendig, um acht verschiedene Arrays zu erzeugen. Eine Lösung mit **x** wäre in diesem Falle wirklich nix:

```
@feld{'A'..'H'} = ([0..7]) x 8;
```

Der Wiederholungsoperator kopiert nur die einmal erzeugte Referenz und alle acht Hash-Schlüsseln würden auf den selben Array zeigen, wie eine schnelle Überprüfung per **say Dumper(%feld);** (**Data::Dumper**) auch zeigt:

```
$VAR1 = {
  'F' => [
           0,
           1,
           2,
           3,
           4,
           5,
           6,
           7
         ],
  'A' => $VAR1->{'F'},
  'E' => $VAR1->{'F'},
  'B' => $VAR1->{'F'},
  'H' => $VAR1->{'F'},
  'C' => $VAR1->{'F'},
  'D' => $VAR1->{'F'},
  'G' => $VAR1->{'F'}
};
```

Ausblick

All das war eine lange Vorrede, weil die Nachrichten nicht mehr länger in einer einfachen Liste (Array) gelagert, sondern nach Themen oder Wichtigkeit sortiert werden sollen. Wer mag, kann versuchen, diese Anforderung allein zu implementieren, bevor in der nächsten Ausgabe die Auflösung erfolgt.

LINKS

[1] http://www.freiesmagazin.de/freiesMagazin-2011-07

[2] https://metacpan.org/

[3] http://prepan.org/

[4] http://perlweekly.com/

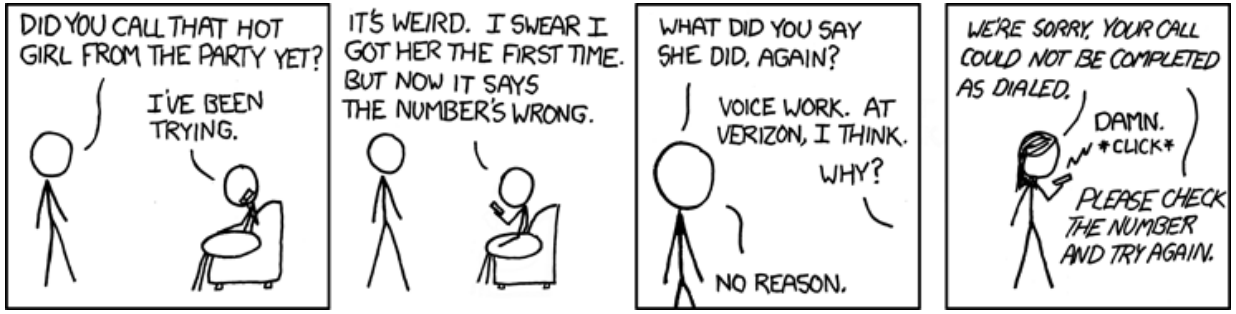
[5] http://perlbuzz.com/

[6] http://perl-tutorial.org/

Autoreninformation

Herbert Breunung (Webseite) ist seit sieben Jahren mit Antworten, Vorträgen, Wiki- und Zeitungsartikeln in der Perlgemeinschaft aktiv. Dies begann mit dem von ihm entworfenen Editor Kephra,

Diesen Artikel kommentieren



„Avoidance“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/615

Python – Teil 10: Kurzer Prozess

von Daniel Nögel

Im letzten Teil der Python-Reihe (siehe freiesMagazin 10/2011 [1]) wurde die `urllib`-Bibliothek vorgestellt und damit ein Client für den FileHoster BayFiles.com implementiert. In diesem Teil soll das `subprocess`-Modul vorgestellt werden, mit dem sich Prozesse starten, Rückgabewerte auslesen und Pipes umsetzen lassen.

Besprechung der Übung

Zunächst soll aber die im letzten Teil vorgeschlagene Aufgabe besprochen werden. Umgesetzt werden sollte eine Download-Funktion mit Fortschrittsanzeige, die es auch erlaubt, begonnene Downloads fortzusetzen.

Eine fertige Beispiellösung findet sich bei GitHub [2]. Einige Besonderheiten sollen im Folgenden besprochen werden.

In Zeile 12 wird die Zieldatei im Modus **`ab`** geöffnet. Dies bedeutet, dass die Datei binär geöffnet wird und neue Daten angehängt werden (**`append`**). Wenn die Zieldatei bereits vorhanden war (**`resume`** in Zeile 11 also den Wert **`True`** zugewiesen bekommen hat), wird in Zeile 15/16 zunächst die aktuelle Dateigröße ermittelt. Dies wäre auch mit der Funktion **`os.path.getsize`** möglich, da die Datei hier aber bereits geöffnet vorliegt, wird mit dem Aufruf **`fh.seek(0, 2)`** die Position auf Byte 0 relativ zum Ende der Datei gesetzt und die Position (und damit die Größe) mit **`fh.tell()`** ermittelt.

Diese Größenangabe wird in den Zeilen 17/18 genutzt, um einen HTTP-Request zu erzeugen, der die restliche Datei vom Server anfordert. Der entsprechende Eintrag im HTTP-Header lautet dann beispielsweise **`Range: bytes=100-`**. Wenn der Fehler **`416`** geworfen wird, bedeutet dies, dass die lokale Datei bereits vollständig ist und der Server keine weiteren Daten bereit hält. In diesem Fall gilt der Download als abgeschlossen (Zeilen 22-25).

In den Zeilen 27-34 wird zunächst die Gesamtlänge des Downloads ermittelt. Durch die Verwendung des Range-Headers liefert der Server auf die Frage nach **`Content-Length`** allerdings lediglich die Restgröße des Downloads zurück, sodass die Größe der bereits heruntergeladenen Daten hinzuaddiert werden muss (Zeile 28). Tritt hierbei ein Fehler auf, wird die Größe pauschal auf **`999999999`** Bytes festgelegt. Eine Lösung, die hier auf Grund der Kürze gewählt wurde. Schöner wäre es beispielsweise, in diesen Fällen die Größe auf **`-1`** zu setzen und weiter unten im Code statt einer Fortschrittsanzeige die Meldung „Gesamtgröße konnte nicht ermittelt werden“ anzuzeigen.

Der Code-Block in den Zeilen 35-41 wird ausgeführt, wenn die Zieldatei noch nicht vorhanden war. Auch hier wird die Download-Größe ermittelt und im Fehlerfall auf **`999999999`** gesetzt.

In den Zeilen 43-49 findet der eigentliche Download statt. Der Ausdruck **`while content`** ist so lange wahr, wie der Aufruf **`wh.read(blocksize)`** in den Zeilen 43 und 46 Daten vom Server zurückliefert und an den Namen **`content`** bindet. Die gelesenen Daten werden mit einem Zähler hochgezählt (Zeile 45) und in die Zieldatei geschrieben (Zeile 46). In den Zeilen 47 und 48 wird schließlich – wie schon im letzten Teil dieser Einführung – eine sehr einfache Fortschrittsanzeige umgesetzt.

Da nicht jeder Server die Anfrage **`Content-Length`** unterstützt und auch nicht jeder Server den Range-Header berücksichtigt, bietet dieses Vorgehen insgesamt durchaus Spielraum für Fehler und Probleme. Dennoch wird durch das Skript veranschaulicht, wie man einen derartigen HTTP-Download mit Fortsetzen-Funktion prinzipiell umsetzen würde.

subprocess

Das **`subprocess`**-Modul [3] ist ein recht mächtiges Werkzeug rund um das Erzeugen von Prozessen. Damit lassen sich nicht nur Prozesse ausführen, sondern auch Rückgabewerte und Fehler auslesen, Eingaben senden und Pipes erzeugen. Es stellt damit eine wichtige Alternative zu der Funktion **`os.system`** dar, die von Anfängern gerne genutzt wird, von der aber sogar in der offiziellen Python-Dokumentation abgeraten wird [4].

Statt also beispielsweise einen neuen Tab im Firefox auf diese Weise zu öffnen

```
>>> os.system("firefox -new-tab www↷
.heise.de")
0
```

wird folgendes **subprocess**-Pendant empfohlen:

```
>>> subprocess.call(["firefox", "-↷
new-tab", "www.heise.de"])
0
```

Beide Funktionen führen das gewünschte Kommando aus und geben den Rückgabewert des Programmes zurück – in diesem Fall jeweils 0. Ein Unterschied fällt aber direkt ins Auge: Der Aufruf von **subprocess.call** erfolgt mittels einer Liste, **os.system** erwartet dahingegen eine einfache Zeichenkette. Dieser Unterschied gilt für die meisten Funktionen im **subprocess**-Modul: Sie erwarten die auszuführenden Kommandos und Argumente als Liste. Zwar lässt sich dies auch umgehen – aus Sicherheitsgründen wird davon aber abgeraten [5]. Mit **shlex.split** [6] gibt es eine Funktion, mit der sich aus einer Zeichenkette bequem die entsprechende Liste erzeugen lässt, so dass die Nutzung von **subprocess** kaum Mehraufwand bedeutet.

Einfache Ausgaben verarbeiten

Nun stellen die beiden Befehle oben sicher einen sehr trivialen Fall im Zusammenhang mit Prozessen dar. Es soll lediglich ein Programm ausgeführt werden, etwaige Ausgaben sind nicht von Interesse. Etwas mehr bietet die Funktion **check_output**. Mit dieser Funktion lässt sich die Ausgabe eines Prozesses auslesen:

```
>>> ret = subprocess.check_output↷
(["wine", "--version"])
>>> print ret
'wine-1.3.32\n'
```

Hier wird die aktuelle Wine-Version durch den Aufruf von **wine --version** ermittelt. Die Ausgabe des Aufrufes wird an den Namen **ret** gebunden. Nun könnte beispielsweise überprüft werden, ob eine bestimmte Wine-Version vorliegt oder eben nicht.

Dies gilt aber nur für den Fall, dass das aufgerufene Programm den Rückgabewert 0 hatte, also fehlerfrei beendet wurde [7]. Andernfalls wird die Ausnahme **CalledProcessError** geworfen. Wird diese abgefangen, lassen sich weitere Informationen über den Fehler, inklusive der Fehlerbeschreibung, ermitteln, wenn das Programm eine solche ausgegeben hat:

```
>>> try:
...     ret = subprocess.check_output(["ls", "/home/nicht-existierender-↷
benutzer"])
... except subprocess.CalledProcessError as inst:
...     print inst.output
...     print inst.returncode
...
ls: Zugriff auf /home/nicht-existierender-benutzer nicht möglich: Datei ↷
oder Verzeichnis nicht gefunden
2
```

check_output eignet sich besonders dann, wenn kleinere Ausgaben schnell durchlaufender Programme ausgelesen werden sollen und im Fehlerfall eine Exception erwartet wird. Das ist nicht immer der Fall: Manche Programme kodieren über den Rückgabewert auch zusätzliche Informationen, die nicht unbedingt auf einen schweren Fehler hinweisen. Da **check_output** weiterhin erst wartet, bis der aufgerufene Prozess beendet wurde, ist es in der Regel auch nicht sonderlich für Programme geeignet, die länger laufen und dabei kontinuierlich Text generieren, der in Echtzeit verarbeitet werden soll.

Das Schweizer Taschenmesser

Für die Verarbeitung derartiger Ausgaben eignet sich die Klasse **Popen**. Sie gewährt Zugriff auf Ein- und Ausgaben in Form von dateiähnlichen Objekten.

```
process = subprocess.Popen(["find",↷
 "/"], stdout=subprocess.PIPE)
while process.poll() is None:
    print process.stdout.readline()
```

Hier wird das Root-Verzeichnis rekursiv durchlaufen und die Ausgabe zeitnah verarbeitet, zusätzlich wird der Parameter **stdout** genutzt. Durch die Übergabe von **subprocess.PIPE** wird er-

reicht, dass die Ausgaben, die das Programm in die Standardausgabe schreibt, durch **Popen** verarbeitet werden. Das **Popen**-Objekt wird hier an den Namen **process** gebunden. Die Methode **poll()** gibt so lange **None** zurück, bis das Programm beendet wurde. Dann gibt es den Rückgabewert des Programmes zurück. Solange der Aufruf von **process.poll()** also **None** liefert, kann davon ausgegangen werden, dass **find** noch läuft. In diesem Fall wird mittels **process.stdout.readline()** jeweils eine Zeile des Ausgabe gelesen und in diesem Fall mit **print** ausgegeben.

Der direkte Zugriff auf **stdout** und **stderr** bringt aber Nachteile mit sich, so besteht etwa die Gefahr sogenannter „Deadlocks“. Außerdem blockiert beispielsweise die Methode **readline()** den Programmfluss so lange, bis tatsächlich eine neue Zeile in den Ausgabepuffer geschrieben wurde. Würde das aufgerufene Programm gar keine Ausgabe in die Standardausgabe schreiben, würde der Programmfluss so lange blockieren, bis das Programm durchgelaufen ist.

Wo es möglich und sinnvoll ist, sollte daher die Methode **communicate()** eingesetzt werden. Diese blockiert zwar ebenfalls, bis das Programm beendet wurde, liefert dafür zuverlässig und ohne die Gefahr besagter Deadlocks die gewünschten Ausgaben. Das hier vorgestellte Vorgehen bietet sich also tatsächlich nur an, wenn die Ausgabe länger laufender Programme zeitnah bearbeitet werden soll und etwaige Seiteneffekte ausgeschlossen werden können.

Communicate und Pipes

In ihrer einfachsten Verwendung ist die Methode **communicate()** eine Alternative zu der Funktion **check_output()**. Zum veranschaulichen:

```
process = subprocess.Popen(["du", ↷
"-s", "/etc", "/nicht-existierender↷
-ordner"], stdout=subprocess.PIPE, ↷
stderr=subprocess.PIPE)
stdout, stderr = process.↷
communicate()
```

Hier soll jeweils die Gesamtgröße der Verzeichnisse **/etc** und **/nicht-existierender-ordner** ermittelt werden. Der Inhalt der Standardausgabe wird dabei an **stdout** gebunden, der Inhalt der Fehlerausgabe an **stderr**.

```
>>> print stdout
10036 /etc

>>> print stderr
du: kann Verzeichnis "/etc/sudoers.↷
d" nicht lesen: Keine Berechtigung
du: Zugriff auf "/nicht-↷
existierender-ordner" nicht möglich↷
: Datei oder Verzeichnis nicht ↷
gefunden
```

Das Programm (**du**) wird also ausgeführt. **process.communicate()**, wartet, bis das Programm beendet wurde, und bindet Standard- und Fehlerausgabe dann an die angegebenen Namen.

Mit **Popen** und **communicate()** lassen sich auch recht einfach Pipes realisieren, wie man sie aus der Shell kennt. So liefert das Programm **ps** für gewöhnlich Informationen über laufende Prozesse. Häufig wird die Ausgabe dieses Programms in der Shell an **grep** weitergeleitet, um bestimmte Prozesse heraus zu filtern:

```
$ ps aux | grep firefox
daniel   7322  0.0  3.3 590272 68920↷
 ?       Sl   15:53   0:02 firefox
daniel   8257  0.0  0.0  12548  1052↷
 pts/8   S+   17:27   0:00 grep ↷
firefox
```

Eine derartige Pipe lässt sich auch mit **subprocess** realisieren:

```
process1 = subprocess.Popen(["ps", ↷
"aux"], stdout=subprocess.PIPE)
process2 = subprocess.Popen(["grep↷
", "firefox"], stdin=process1.↷
stdout, stdout=subprocess.PIPE)
stdout, stderr = process2.↷
communicate()
```

Hier sind zwei **Popen**-Instanzen nötig. Die Eingabe der zweiten Instanz **process2** ist die Ausgabe der ersten Instanz **process1**. Das ließe sich so prinzipiell beliebig erweitern. So wäre beispielsweise folgende Kommandozeile denkbar, die direkt die PIDs der mittels **ps aux | grep firefox** gefundenen Prozesse ermittelt:

```
$ ps aux | grep firefox | awk '{↷
print $2}'
7322
8826
```

Das Pendant in Python sähe wie folgt aus:

```
process1 = subprocess.Popen(["ps", "aux"], stdout=subprocess.PIPE)
process2 = subprocess.Popen(["grep", "firefox"], stdin=process1.stdout, ↷
stdout=subprocess.PIPE)
process3 = subprocess.Popen(["awk", "{print $2}"], stdin=process2.stdout, ↷
stdout=subprocess.PIPE)
stdout, stderr = process3.communicate()
```

Zusammenfassung und Einschränkung

subprocess ist ein mächtiges Werkzeug, wenn es darum geht, externe Programme zu starten und deren Ausgabe auf verschiedene Weise zu verarbeiten. Dabei sollte nach Möglichkeit darauf verzichtet werden, direkt die dateiähnlichen Objekte **stdout**, **stderr** und **stdin** zu verwenden. Die Methode **communicate()** sowie die Funktionen **check_output()** und **call()** bieten in der Regel verlässliche Alternativen, abhängig vom konkreten Anwendungsfall.

Der gerne genutzte **shell**-Parameter, den **Popen** und **check_output()** kennen, wurde hier nicht besprochen. Er erlaubt es, Befehle wie **ps aux | grep firefox** von **subprocess** durch eine Shell ausführen zu lassen. So stehen zwar einige erweiterte Funktionen der Shell zur Verfügung und auch das aufwändige Nachbilden von Pipes bleibt dem Nutzer erspart; gerade wenn Teile der Kommandozeile aber aus fragwürdigen Quellen zusammengesetzt werden – etwa mit Zeichenketten von einer Webseite – kann dies aber zu einer ernsthaften Sicherheitslücke werden [8].

Schließlich sei noch darauf hingewiesen, dass **subprocess** sehr gut geeignet ist, wenn Programme angesprochen werden sollen, für die es keine Python-Anbindung gibt. Soll beispielsweise das Kodierungsprogramm MEncoder aus einem Python-Skript heraus aufgerufen werden, biete sich **subprocess** an. In sehr vielen Fällen ist es aber besser, die Funktionalität einfacherer Programme in Python nachzubilden oder bestehende Anbindungen zu nutzen.

Statt also **find** aufzurufen, empfiehlt es sich, ein Verzeichnis mit **os.walk** rekursiv zu durchlaufen. Statt die Größe einer Datei mit **du** zu ermitteln, sollte **os.path.getsize** verwendet werden. Und statt **ftp** mit **subprocess** „fernzusteuern“, sollte das Python-Modul **ftplib** genutzt werden.

Externe Kommandos und Programme können in vielerlei Hinsicht zu schwer nachzuvollziehbaren Fehlern führen. Die Programme sind unter Umständen gar nicht installiert oder nicht in einem Verzeichnis abgelegt, das in der Umgebungsvariable **PATH** gelistet wird. Unter Umständen gibt es unterschiedliche Varianten von Programmen mit dem selben Namen, wie etwa **netcat**, das es in mehreren unterschiedlichen Implementierungen gibt. Vielleicht verhält sich ein Programm auch auf verschiedenen Plattformen unterschiedlich, erwartet Parameter unter Linux nach Bindestrichen, unter Windows nach Schrägstrichen.

Aus diesen Gründen sollte das **subprocess**-Modul mit Bedacht eingesetzt werden. In der Regel lohnt sich eine kurze Suche nach einer Python-Bibliothek mit der gewünschten Funktionalität.

LINKS

[1] http://www.freiesmagazin.de/freiesMagazin-2011-10

[2] https://github.com/jbarabbas/Simple-download-function/blob/master/download.py

[3] http://docs.python.org/library/subprocess.html#subprocess-replacements

[4] http://docs.python.org/library/os.html#os.system

[5] http://docs.python.org/library/subprocess.html#security

[6] http://docs.python.org/library/shlex.html#shlex.split

[7] http://en.wikipedia.org/wiki/Exit_status

[8] http://docs.python.org/library/subprocess.html#frequently-used-arguments

Autoreninformation

Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

Diesen Artikel kommentieren

ArchivistaVM – Server-Virtualisierung für die Hosentasche mit USB-Stick

von Urs Pfister

Dieser Artikel stellt einen portablen KVM-Server mit *ArchivistaVM* vor, der von einem USB-Stick auf jedem Rechner in ca. 30 Sekunden gestartet und in Betrieb genommen werden kann. Dabei muss keine Software installiert werden, die Software zur Virtualisierung (KVM), ein X-Server und Web-Server sowie das GUI (Web-Browser) werden komplett in den Hauptspeicher geladen. Die Gäste werden entweder direkt vom USB-Stick oder von der Festplatte gestartet. Nach dem Starten kann lokal (X-Server mit Web-Browser) oder von einem beliebigen Rechner aus (Web-Browser mit Java-Applets) gearbeitet werden.

Eine Demo, die das Resultat innerhalt weniger Sekunden vorführt, kann man unter [1] finden. Weiter seien an dieser Stelle einige Vorteile von *ArchivistaVM* angeführt:

1. In fast jedem Haushalt, in jedem Büro, fast an jeder Ecke gibt es Rechner, die eben mal stillstehen.
2. Jeder Rechner und viele Notebooks bringen heute zumindest eine DualCore-CPU und 2 GByte RAM mit, was ausreicht. Wirklich Spaß macht die Lösung ab einer QuadCore-CPU und 4 GByte RAM.
3. Ein virtualisiertes OS soll zur Datensicherung gestartet werden. Weshalb ein System aufspielen, Daten zurückspielen, um den Job erledigen zu können, wenn die Lösung in 30 Sekunden mit einem USB-Stick und der Sicherungsplatte zu haben ist?
4. Ein KVM-Server vom USB-Stick im RAM arbeitet extrem schnell. Dadurch, dass die Software ins RAM installiert wird, dauert das Starten nicht länger, als wenn von Festplatte gestartet würde.
5. Geht die Virtualisierung in der Wolke (Cloud) nicht viel einfacher? Spätestens wenn die Wolke mal das Backup vergisst, ist die Einfachheit (inkl. Daten) weg.

Nachfolgend wird aufgezeigt, wie ein USB-Stick mit *ArchivistaVM* erstellt wird und wie mit dem System gearbeitet werden kann. Weiter werden die wichtigsten Optionen für automatisierte Setups vorgestellt. Um diesen Artikel durchzuarbeiten, wird ein USB-Stick mit mindestens 8 GByte und ein Debian/Ubuntu (oder *ArchivistaVM* ab CD) benötigt.

USB-Stick erstellen

Nachfolgend wird das Erstellen eines USB-Sticks oder einer CD-ROM unter Debian/Ubuntu beschrieben. Die ca. 350 MB große ISO-Datei kann unter [2] herunterladen werden.

KVM-Server auf CD-ROM

Wem die Installation auf einem USB-Stick zu kompliziert ist, der kann dennoch mit *ArchivistaVM* arbeiten, denn der portable KVM-Server funktioniert auch auf CD-ROM. Allerdings passen CD-Laufwerke nicht in die Hosentasche ...

Sofern mit einer CD-ROM gearbeitet wird, kann diese nach dem Brennen direkt gestartet werden. Dazu ist beim Starten wie untenstehend dargestellt `ram` einzugeben:

Startbildschirm von ArchivistaVM.

Mit Drücken der Enter-Taste wird das Setup von *ArchivistaVM* durchgeführt. Der Vorgang von CD-ROM dauert etwas länger, mehr als eine Minute dürfte aber nicht notwendig sein, um *ArchivistaVM* zu starten. Nach dem Starten kann die CD-ROM entfernt werden. Die CD wird für das weitere Arbeiten nicht mehr benötigt.

Hinweis: Ohne die Option `ram` wird das Setup-Programm von *ArchivistaVM* gestartet, um das System auf der Festplatte einzurichten. Spätestens bei der Mitteilung, dass alle Daten zerstört werden, sollte hier auf „*Nein*“ geklickt werden ...

Einrichten eines USB-Stick

Nach dem Download unter Debian/Ubuntu ist eine Konsole (xterm) zu öffnen und mit **`sudo su`** – wird zu root gewechselt. Zunächst werden die Pakete lilo und syslinux benötigt. Diese

können mit **apt-get install lilo syslinux** installiert werden. Danach kann der USB-Stick eingelegt werden. Wichtig dabei ist, die richtige Adresse des Sticks zu finden. Dazu kann **dmesg** verwendet werden.

```
# dmesg
usb 7-3: new high speed USB device using ehci_hcd and address 15
scsi20 : usb-storage 7-3:1.0
scsi 20:0:0:0: Direct-Access     USB      Flash  DISK     1100 PQ: 0 ANSI: 0
CCS
sd 20:0:0:0: Attached scsi generic sg5 type 0
sd 20:0:0:0: [sdb] 1957888 512-byte logical blocks: (1.00 GB/956 MiB)
sd 20:0:0:0: [sdb] Write Protect is off
sd 20:0:0:0: [sdb] Mode Sense: 43 00 00 00
sd 20:0:0:0: [sdb] Assuming drive cache: write through
sd 20:0:0:0: [sdb] Assuming drive cache: write through
sdb: sdb1
sd 20:0:0:0: [sdb] Assuming drive cache: write through
sd 20:0:0:0: [sdb] Attached SCSI disk
```

Auf den zuletzt angezeigten Zeilen findet sich die Adresse des Sticks. Nun kann der Stick mit **cfdisk** vorbereitet werden, in diesem Beispiel ist **cfdisk /dev/sdb** zu verwenden.

Hinweis: Die nachfolgend verwendeten Einstellungen (**/dev/sdb**) können übernommen werden, wenn genau eine Festplatte im System vorhandenn ist und der USB-Stick als zweites Gerät (**sdb**) erscheint. Bei zwei internen Festplatten wäre **/dev/sdc** zu verwenden, bei drei Platten **/dev/sdd** und so weiter. Wird eine falsche Gerätenummer verwendet, wird ein Datenverlust zu beklagen sein – und der USB-Stick für *ArchivistaVM* wird trotzdem nicht eingerichtet.

Zurück zum Stick, dieser muss bootbar sein und ist mit FAT16 (Typ 06) zu formatieren. Weiter sind drei Partitionen einzurichten:

- Partition Fat 16 (Typ 06) mit 1024 MByte
- Partition Swap (Typ 82) mit 2048 MByte (empfohlen)
- Partition ext3 (Typ 83) mit dem übrigem Platz

Die nachfolgende Abbildung verdeutlicht dies:

*Einrichten der Festplatte mit **cfdisk**.*

Nachdem der Stick initialisiert wurde, können über die Konsole die restlichen Schritte durchgeführt werden:

```
# mkdosfs /dev/sdb1
# lilo -M /dev/sdb
# syslinux /dev/sdb1
# mkdir /in
# mkdir /out
# mount -o loop avvm-64bit.iso /in
# mount /dev/sdb1 /out
# cp /in/* /out
# umount /in
# umount /out
```

Nun sollte der Stick booten und der KVM-Server gestartet werden. Vorher sollte noch eine Daten-Partition eingerichtet werden. Dies wird mit **mkfs.ext4 /dev/sdb3** erreicht.

Ob es sich lohnt, eine Swap-Partition auf einen USB-Stick zu legen, muss anhand der Qualität des Sticks entschieden werden. Bei preisgünstigeren Modellen sollte es eher nicht gemacht werden. Die Swap-Partiton wird mit **mkswap /dev/sdb2** eingerichtet. Mit oder ohne Swap, der USB-Stick steht nun zum Arbeiten bereit.

Arbeiten mit dem KVM-Server

Beim Starten ist zu beachten, dass der RAM-Modus aktiviert werden muss. Dazu ist beim ersten Start-Bildschirm mindestens **ram** einzugeben.

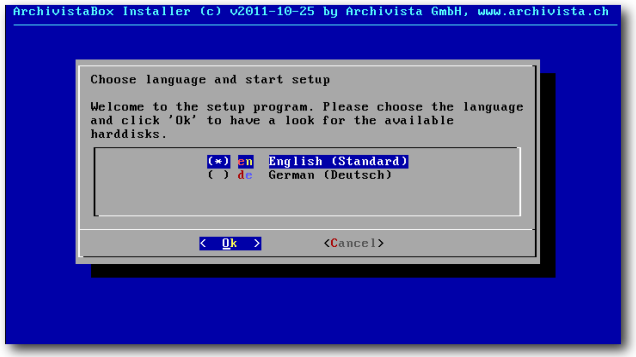
Hinweis: Ohne die Option **ram** wird das Setup-Programm von *ArchivistaVM* gestartet, um das System auf der Festplatte einzurichten. Wer dabei unbedacht auf „*Ja*“ klickt, riskiert, dass das Setup-Programm die Festplatte zerstört.

Booten des Servers mit `ram ramdisk`

Bei diesen beiden Optionen wird der USB-Stick so gestartet, dass die virtualisierten Instanzen direkt auf der dritten Partition des USB-Sticks gespeichert werden. Das gesamte System (KVM und Images) liegt auf dem gleichen Stick, es kann hier vom KMV-Server für die Hosentasche gesprochen werden. Damit einzig mit dem USB-Stick gearbeitet werden kann, ist beim ersten Prompt wie nachfolgend dargestellt **ram ramdisk** einzugeben:

Booten von ArchivistaVM im RAM.

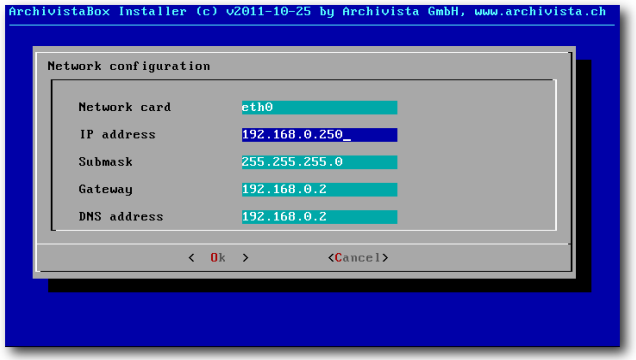
Das System wird nun eingerichtet. Nach einigen Sekunden startet das Setup-Programm. Die erste Abfrage ist zu bestätigen.



Auswahl der Sprache im Setup.

Nach zwei bis drei Sekunden erscheint eine Liste mit den vorhandenen Festplatten. Es kann dabei irgendeine Festplatte ausgewählt werden. Im Modus **ram ramdisk** erfolgt kein Zugriff auf die internen Festplatten. Es wird einzig auf diese Platte zurückgegriffen, sofern entweder eine CD-ROM oder die Partitionen auf dem USB-Stick nicht eingebunden werden können (z. B. weil sie gar nicht vorhanden sind).

Nach weiteren 10 bis 15 Sekunden erfolgt eine Abfrage für die gewünschten Netzwerk-Einstellungen. *ArchivistaVM* verlangt nach einer fixen IP-Adresse (ein Server für die Virtualisierung ist nicht unbedingt sinnvoll ohne feste IP-Adresse). Dabei sind die Daten analog zur Abbildung einzugeben:

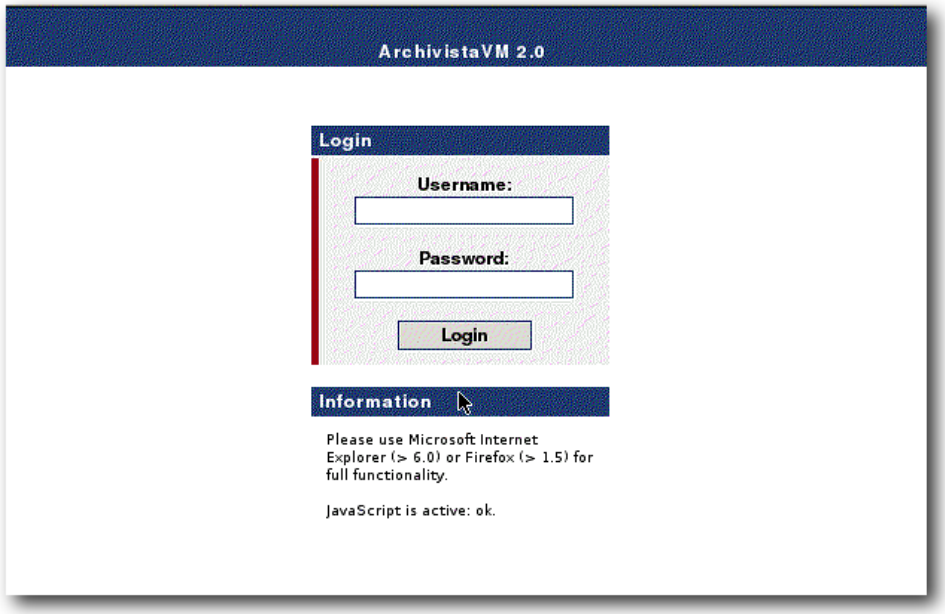


Einstellen der Netzwerkkonfiguration.

Hinweis: Sollte keine eindeutige Netzwerkadresse vorhanden sein, so kann beim Computer das Netzwerkkabel abgehängt werden. Dann kann die voreingestellte Adresse 192.168.0.250 problemlos verwendet werden. Ein Arbeiten mit *ArchivistaVM* ist auch ohne Netzanschluss lokal möglich.

Nach weiteren ca. 5 Sekunden erscheint eine Meldung, dass das System nun zum Arbeiten zur Verfügung steht (System bitte nicht neu starten, das wäre nicht im Sinne einer RAM-basierten Installation). Es kann nun entweder mit einem externen Rechner per Web-Browser auf *ArchivistaVM* zugegriffen werden oder durch Eingabe von **go** (mit Enter-Taste) ein X-Server gestartet werden.

Nach etwa 2 Sekunden kann die Tastatur festgelegt werden. Danach wird umgehend der Browser gestartet; zum Anmelden ist beim Benutzer **root** und beim Passwort **archivista** einzugeben:



Nun kann man sich nun im Browser als Benutzer anmelden.

Damit ist die Installation vollständig. Eine Anleitung zum Arbeiten mit dem KVM-Server alias *ArchivistaVM* findet man unter [3].

Start mit interner/externer Festplatte

Wer mit einer vorbestimmten internen/externen Festplatte arbeiten möchte, kann mit den Parametern **ramswap./dev/xxx** und **ramdata./dev/yyy** sowohl die gewünschte Swap- als auch Daten-Partition bestimmen. Dazu ein Beispiel:

```
ram ramswap./dev/sdc2 ramdata./dev/sdd1
```

Mit den obenstehenden Einstellungen wird die Swap-Partition auf der dritten Platte und der zweiten Partition gestartet, die Daten-Partition erfolgt ab der vierten Festplatte und der ersten Partition.

Start und Formatieren von Festplatten

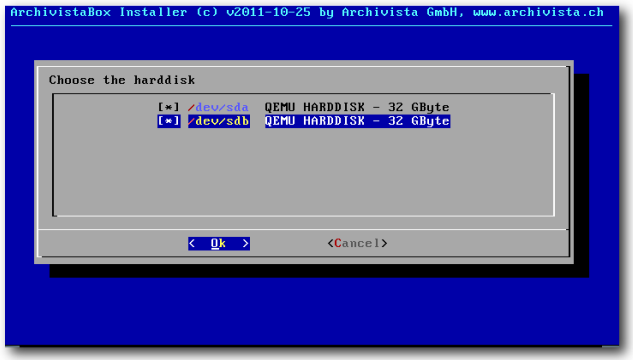
Mit der Option **ram formathd** kann (bei leeren Festplatten) erzwungen werden, dass eine beim Start gewählte Platte formatiert und für das Arbeiten mit *ArchivistaVM* eingerichtet wird.

Hinweis: Mit **ram formathd** können nur leere Festplatten eingerichtet werden. Bereits formatierte Platten werden nicht neu eingerichtet. **ram formathd** sollte dennoch vorsichtig eingesetzt werden. Die Entscheidung, ob formatiert wird oder nicht, erfolgt aufgrund der Partitionstabelle. Sollte diese defekt sein, so wird die Platte formatiert; auch wenn sich eventuell noch gewisse Daten restaurieren ließen.

Arbeiten mit Software-Raid

Virtualisierung ist nicht nur eine Frage der Prozessoren oder des Hauptspeichers, in erster Linie wichtig für die Geschwindigkeit ist der Zugriff auf die Festplatten.

Mit der Option **ram formathd** kann bei leeren Festplatten auch ein Software-Raid eingerichtet, sofern beim Auswählen der Festplatten mehrere Festplatten ausgewählt werden:



Auswahl der Festplatte für das Software-Raid.

Bei zwei Festplatten wird Raid1 eingerichtet, bei vier, sechs oder noch mehr Platten erfolgt das Einrichten mit Raid10. Beim Einrichten eines Raids erfolgt eine Bestätigungsabfrage. Soll Raid0 eingerichtet werden, muss bereits beim Starten **ram formathd raid0** eingegeben werden. An dieser Stelle sei ausdrücklich darauf hingewiesen, dass Raid0 auf eigene Gefahr verwendet wird.

Hinweis: Derzeit können nur von *ArchivistaVM* eingerichtete Software-Raids verwendet werden. Allerdings könnte ein Raid auf der Konsole nachträglich von Hand zugeschaltet werden. Die Treiber aller gängiger Hardware-Raid-Kontroller sind ebenfalls enthalten, wobei *ArchivistaVM* nicht mit sämtlichen Karten getestet werden konnte. Allenfalls müssten diese von Hand geladen werden, damit die Partitionen eingebunden werden können.

Alternativ zum direkten Einrichten eines Raid-Verbundes im RAM-Speicher kann das System zunächst auch komplett auf die Festplatte gespielt werden. Dies erfordert zwar einen Neustart, dafür kann *ArchivistaVM* danach wahlweise mit oder ohne RAM-Modus eingesetzt werden. Um *ArchivistaVM* auf der Festplatte einzurichten, ist beim Starten auf die Eingabe der Option **ram** zu verzichten. Selbstverständlich ist beim zweiten Start die Option **ram** einzugeben, da ansonsten das System von der Festplatte gestartet wird.

Automatisierung

Der Installer von *ArchivistaVM* wurde in Perl (objektorientiert) realisiert. Das Anpassen des Setup-Programmes dürfte für einigermassen kundige Perl-Programmierer keine grossen Schwierigkeiten darstellen. Bevor jemand die Datei **initrd.img** und das Setup-Programm **install.pl** selber anzupassen gedenkt, sei auf die bereits vorhandenen Start-Optionen verwiesen.

Arbeiten mit dem Auto-Modus

ArchivistaVM arbeitet beim Booten mit Syslinux bzw. ab CD-ROM mit Isolinux. Dazu existieren die beiden Dateien **syslinux.cfg** sowie **isolinux.cfg** direkt im Hauptverzeichnis der ISO-Datei bzw. der ersten Partition des USB-Sticks.

Einige Informationen zu dieser Datei. Zunächst wird beim Booten zehn Sekunden gewartet, bis der Start erfolgt. Dies ist etwas missverständlich, aber abgerechnet wird in Zehntel-Sekunden. Wenn man eine Minute warten möchte, dann wäre **TIMEOUT** auf **600** zu setzen. Mit **DEFAULT** wird festgelagt, welche Definition ohne weitere Eingaben gestartet wird. Wenn man z. B. erreichen möchte, dass immer im RAM-basierten Modus gestartet wird, muss bei Default **ram** festgelegt werden.

```
DEFAULT linux
TIMEOUT 100

PROMPT 1
DISPLAY boot.msg

label linux
kernel vmlinuz
APPEND initrd=initrd.img quiet ramdisk_size=81920

label ram
kernel vmlinuz
APPEND initrd=initrd.img quiet ram ramdisk_size=2097152

label box244
kernel vmlinuz
APPEND initrd=initrd.img quiet ram ramdisk_size=4194304
  keyboard.de_CH lang.de auto ip.192.168.2.244
  submask.255.255.255.0 gw.192.168.2.1 dns.192.168.2.1 ramdisk
```

Listing 1: syslinux.cfg

Spannend ist weiter die Variante **box244**. Dabei handelt es sich um ein automatisiertes Startup, welches das System automatisiert hochfährt. Zentral ist dabei **auto**. Damit wird erreicht, dass das Setup-Programm keine Fragen mehr stellt. Und in diesem Modus ist das System in 30 Sekunden auf einem USB 2.0-Stick eingerichtet.

Es gibt weitere Parameter, die man direkt im Setup-Programm **install.pl** unter der Funktion **sub cmdline** findet.

Telnet-Zugriff bei Problemen

Am Ende des Setups wird ein SSH-Server gestartet, sodass auf *ArchivistaVM* problemlos auf der Konsole zugegriffen werden kann. Was aber, wenn das System vor dem Starten des SSH-Servers hängenbleibt? In einem solchen Falle kann mit **ram** das Setup-Programm gestartet werden. Anstatt den Anweisungen danach Folge zu leisten, wird das Programm abgebrochen. Nun kann auf der Konsole die Netzwerkkarte und ein telnet-Server eingerichtet werden. Dies geht so:

```
ifconfig eth0 192.168.0.250
route add default gw 192.168.0.2
telnetd -l /bin/login
```

Danach kann von einem entfernten Rechner aus mit telnet auf die Konsole zugegriffen werden, beim Anmelden ist das Passwort **archivista** zu verwenden.

Anpassungen an initrd.img

In der Datei **initrd.img** befindet sich das gesamte Mini-Linux. Beim Hochfahren des Systems wird zunächst dieses Mini-System eingerichtet, ehe über den Stick (bzw. CD-ROM) auf **system.os** zugegriffen wird. Die Datei **system.os** enthält ein komplettes Debian-System bzw. *ArchivistaVM*. Beim Hochfahren des Servers wird in einem zweiten Schritt dieses System im RAM eingerichtet, und zwar äußerst schnell. Pro 1 GByte an Software sind etwa 10 Sekunden Zeit notwendig. Beide Dateien können beliebig angepasst werden. **initrd.img** wird wie folgt ausgepackt:

```
$ zcat initrd.img | cpio -iv
```

Dabei wird das gesamte System im aktuellen Verzeichnis ausgepackt. Es können nun die gewünschten Änderungen am System vorgenommen werden. Für das Neupacken des Systems in eine Image-Datei ist der folgende Einzeiler zu verwenden:

```
$ find . | cpio -o -H newc | gzip -↷
c >../isolinux/initrd.img
```

Anpassungen an system.os

In der Datei **system.os** befindet sich das gesamte Linux-System von *ArchivistaVM*. Dazu ist zu sagen, dass bei *ArchivistaVM* nicht Debian- um Debian-Paket installiert wird (dieser Vorgang würde viel zu lange dauern), sondern dass das gesamte System mit **unsquashfs** entpackt wird.

```
$ unsquashfs -dest /os /tmp/cd/↷
system.os
```

Das Packen wird mit dem folgenden Befehl erreicht:

```
$ mksquashfs / /inst/system.os -↷
noappend -e /inst /proc /sys
```

Fazit

Das System läuft unter einer Bedingung, dass die System-Partition des RAM-basierten Systems nicht vollläuft, sehr stabil. Ist das RAM erst mal aufgebraucht, ist die RAM-Platte nämlich unbrauchbar. Ausgeschlossen werden kann es derzeit nicht. Der hier vorgestellte KVM-Server kann jedenfalls mit der gleichen Geschwindigkeit von einem USB-Stick gestartet werden, als wenn das System von der Platte gestartet würde.

Für einen produktiven Server-Betrieb (KVM-Virtualisierung) sollten zwei weitere Dinge beachtet werden. Zum einen ECC-RAMs (bei AMD-Prozessoren laufen die normalen ECC-RAMs problemlos auf jedem Desktop-Board), zum anderen sollte ein Notstrom-Gerät (USV) zur Verfügung stehen. Beide Punkte sind aber bei einem Server-Betrieb ohnehin zu empfehlen.

Neben *ArchivistaVM* existieren auch ArchivistaDMS (Dokumenten-Management) sowie ArchivistaDesktop (DMS und Desktop-Applikationen). Alle drei Systeme enthalten den KVM-Server für die Virtualisierung, und alle drei ISO-Dateien können auf den Stick gepackt und wie obenstehend im RAM direkt gestartet werden. Bei ArchivistaDMS sind mindestens 4 GB RAM erforderlich, beim ArchivistaDesktop sind ca. 8 GB für ein Arbeiten vorzuhalten. Dafür können direkt im RAM OpenOffice, Gimp, Scribus, Kile und viele weiteren Desktop-Applikationen gestartet werden.

LINKS

[1] http://www.archivista.ch/avvm11.gif

[2] http://www.archivista.ch/avvm-64bit.iso

[3] http://213.160.42.154/d2/help/node71.html

Autoreninformation

Urs Pfister (Webseite) kaufte sich mit 16 und dem ersten Lohn einen CPC464, gründete mit 30 die eigene Firma „Archivista", und ist kurz darauf zu Linux hinübergeschwenkt. Seit dieser Zeit gilt seine Leidenschaft Open Source und allem, was dazugehört. Zur Zeit beschäftigt ihn die Weiterentwicklung der ArchivistaBox.

Diesen Artikel kommentieren

BeamConstruct – Linux in der Laserindustrie

von Hans Müller

Das bereits in vorangegangenen Artikeln vorgestellte OpenAPC-Softwarepaket (siehe „Heimautomatisierung für Hardwarebastler Teil 1–3“, freiesMagazin 01/2011 [1], 03/2011 [2] und 06/2011 [3]) hat mit der vor kurzem neu veröffentlichten Version 2 – welche auf den klangvollen Codenamen Aurora Borealis hört – einige umfassend neue Funktionalitäten erhalten, die einen etwas genaueren Blick auf diese Veränderungen rechtfertigen. Neben verschiedenen kleinen Detailverbesserungen, neuen Plug-Ins, die zusätzliche Hardware unterstützen und einer kleineren Umorganisation des gesamten Paketes sticht eine Änderung deutlich heraus: mit der Software *BeamConstruct* ist jetzt eine Applikation verfügbar, welche auf die Ansteuerung von Laserscannersystemen [4] und generell auf Lasermarkieroperationen hin optimiert ist.

Laserscanner?

Der Name indes ist für den unbedarften Benutzer etwas irritierend. Ist ein Scanner normalerweise ein Gerät, mit dem sich Dinge aus der realen Welt in digitale Informationen wie z. B. ein Bild oder ein 3D-Modell umwandeln lassen, so bezeichnet er hier etwas, was eine Oberfläche mit einem Laser abtastet. Das heißt, am Ende einer Bearbeitung mit einem solchen Laserscannersystem steht keine Datei und keine Datensammlung, vielmehr wurde das abgetastete Werkstück selbst verändert. Abhängig von der gewählten Laserleistung, der Geschwindigkeit des Bearbeitungsprozesses und des bearbeiteten Materials können dabei die unterschiedlichsten Aufgaben gelöst und Veränderungen am Material bewirkt werden. Das beginnt beim Abtragen von Oberflächen (z. B. um ein Material zu reinigen), geht weiter bei Lasergravierarbeiten (z. B. zum Beschriften und zur Fertigung von Schildern) sowie beim Laserschweißen und hört beim Laserschneiden noch lange nicht auf.

Solcherlei mit Lasern bearbeiteten Materialien sind dabei mittlerweile so weit verbreitet, dass man es manchmal kaum glauben mag, wofür diese Technik überall verwendet wird. Da ist beispielsweise die Handytastatur, deren Zahlen mit einem Laser erzeugt wurden, das Verfallsdatum oder der Gewinnspielcode auf Lebensmittelverpackungen oder Getränkeflaschen, die matt glänzende, „gebürstete“ Metalloberfläche hochwertiger Geräte oder aber auch die Grillstreifen auf manchen Fertig-Lebensmitteln.

Aber auch für andere Produktionsprozesse kommen Laser zum Einsatz, so beispielsweise bei der Herstellung von Solarzellen.

Die dafür eingesetzten Laserscannersysteme besteht dabei aus einem Scankopf [5], welcher zwei Spiegel beinhaltet, die den Laser in X- und Y-Richtung ablenken, dem Laser selbst und einer Controller-Elektronik, die Scankopf, Laser und die ansteuernde Software miteinander verbindet. Optional kann auch noch ein weiteren Element in der Strahlführung des Lasers vorkommen, welches die Position des Laserfokus verändert. Das ergibt die Möglichkeit, in Z-Richtung zu positionieren. Anwendungsgebiete hierfür sind das Rapid Prototyping [6] oder das In-Glass-Marking, bei dem dreidimensionale Objekte in einem Glasquader als solche sichtbar werden.

Linux!

Und genau an dieser Stelle kommt die jetzt neu im OpenAPC-Paket enthaltene Software *BeamConstruct* zum Einsatz. War es bisher so, dass einzelne Hersteller von Scannercontrollern Treiber für Linux anboten (z. B. Scanlab [7] für alle Karten ab der RTC3) oder aber generell ein plattformunabhängig zu benutzendes TCP/IP-Interface in diese implementiert haben (z. B. Raylase [8] in der SP-ICE2), so sah es bei der Anwendersoftware bisher schlecht aus. Lasermarkierapplikationen gab es nur für Windows; wer Linux benutzen wollte, konnte sich nur selbst etwas programmieren. Das ändert sich mit *BeamConstruct* jetzt grundlegend, hier ist erstmalig eine vollständige und umfassende, für Laserscannersysteme optimierte CAD-Applikation verfügbar, welche auf das wxWidgets-Toolkit [9] aufsetzt und deswegen für Windows und Linux zu haben ist. Unter Linux werden dann natürlich nur die Scannercontroller unterstützt, welche von Haus aus bereits Linux-Unterstützung bieten, allerdings ist das in dem Fall keine echte Ein-

schränkung: der Marktführer Scanlab ist bei diesen dabei und wenn sich die Software entsprechend ihres Potenzials durchsetzt, werden die anderen Hersteller früher oder später zwangsläufig folgen müssen.

Erste Schritte

Nach dem ersten Start von *BeamConstruct* zeigt sich dem Benutzer eine Oberfläche, die dem ebenfalls im OpenAPC-Paket enthaltenen CNConstruct sehr ähnlich ist. Tatsächlich scheinen beide Applikationen eine gemeinsame Basis zu nutzen. Das bietet sich auch an, da beide dem Benutzer ähnliche CAD-Funktionalitäten bieten, welche benötigt werden, um Prozessdaten zu erzeugen. Die Unterschiede beginnen bei deren Nutzung: Können die CNConstruct-Daten verwendet werden, um beispielsweise einen XY-Tisch und eine Fräse anzusteuern, so werden sie bei *BeamConstruct* eben zur Ansteuerung eines Scankopfes und eines Lasers verwendet. Grundsätzlich gilt die folgende Beschreibung aber in nahezu identischer Weise auch für CNConstruct.

Wer ein Laserscannersystem sein Eigen nennt und dieses mit *BeamConstruct* nutzen möchte, sollte die Applikation zuerst entsprechend der eigenen Hardware konfigurieren. Dazu findet sich im Menü „*Project*“ ein Untermenü „*Project Settings ...*“. Dieses öffnet das Fenster für globale Konfigurationen, im Panel „*Hardware*“ kann dann festgelegt werden, welche Geräte von der Software angesteuert werden können. So ist es zum einen möglich, aus einer Liste verfügbarer Treiber für Scannercontroller den passenden Typ auszuwählen und diesen anschließend mit einem Klick auf den Button „*Configure*“ entsprechend den eigenen Wünschen anzupassen. Nach dem gleichen Prinzip können bis zu zwei Motoren ausgewählt und eingestellt werden, mit denen es möglich ist, zusätzliche Bewegungen auszuführen. Damit lassen sich später komplexe Abläufe und Steuerungen realisieren wie beispielsweise das Bearbeiten eines Ringes, welcher dann Schritt für Schritt weiter gedreht wird, das Bearbeiten übergroßer Flächen, welche per zusätzlichem XY-Tisch unter dem Scanner weiter bewegt werden und vieles andere mehr.

An dieser Stelle sollte eine Warnung ausgesprochen werden: Die Laserbearbeitung ist ein gefährlicher Spaß. Kann ein unbedarfter Amateur mit einem vergleichsweise harmlosen Werkzeug wie einer Bohrmaschine beim unsachgemäßen Umgang schon für nennenswerte Schäden und ernsthafte Verletzungen sorgen, so sind Laser um ein vielfaches gefährlicher! Bereits Laserleistungen nur wenig oberhalb von 1 mW (entsprechend Laserklassen ab 3 [10]) können zum sofortigen Erblinden führen, bei entsprechend höheren Leistungen sind auch schwere Verbrennungen und Verletzungen möglich. Sicherheit bietet hier noch nicht einmal die Verwendung von handelsüblichen Laserpointern, da viele Billigprodukte ebenfalls eine viel zu hohe und hierzulande damit eigentlich nicht zulässige Leistung abgeben. Beim Experimentieren mit dieser Technologie ist auf jeden Fall äußerste Vorsicht geboten, die Einhaltung der Sicherheitsbestimmungen und die Einrichtung eines umfassenden Laserschutzes sind also unabdingbar!

Auch ist es keine Lösung, sich einfach nur vom Strahlweg des Lasers fern zu halten. Spätestens wenn dieser auf ein Objekt trifft, wird dessen Strahlung auch reflektiert. Hat das Objekt dann keine exakt ebene Oberfläche (wie es nur bei einem Spiegel der Fall wäre) ist die Reflexionsrichtung und Streuung der Laserstrahlung nicht vorhersehbar. Und auch wenn das Internet voll von Videos ist, in denen Leute scheinbar ohne Folgen völlig planlos mit Lasern herumhantieren, so ist das kein Grund für eine Entwarnung: wenn diese blind sind, sind sie sicher kaum noch in der Lage ein Video über diesen Zustand zu drehen und es bei YouTube hochzuladen.

Sind alle Maßnahmen getroffen, welche dafür sorgen, dass der Laser keinen Schaden anrichten kann, so ist auch die weitere Verwendung von *BeamConstruct* sorgenfrei möglich. Wurden alle Einstellungen vorgenommen, kann das erste Laserprojekt erzeugt werden.

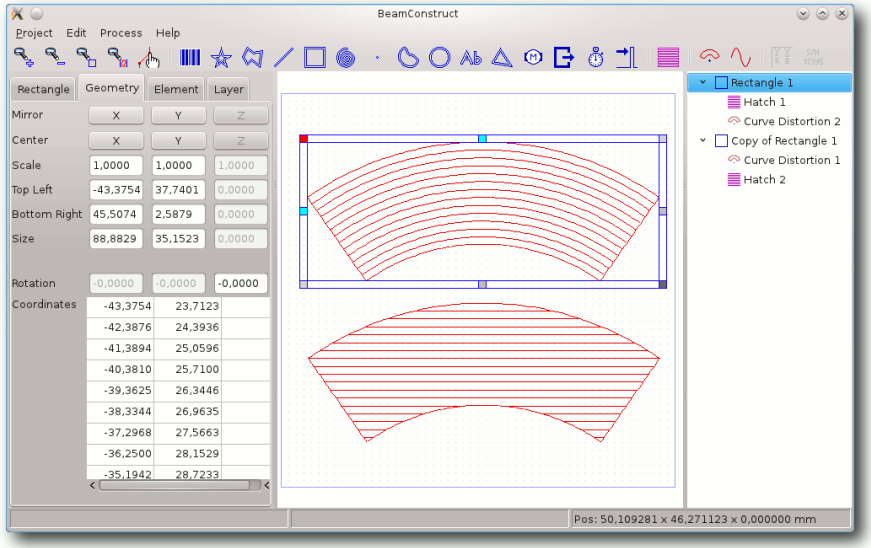
Geometrien und Hierarchien

So finden sich in der Werkzeugleiste am oberen Fensterrand verschiedene geometrische Grundformen wie z. B. ein Kreis, ein Rechteck, ein Dreieck aber auch ein Symbol für Text, für Barcodes und anderes mehr. Wird eines dieser Toolbar-Elemente selektiert, so kann es anschließend im Zeichenbereich erzeugt werden. Am Beispiel des Kreises sieht das so aus, dass ein erster kurzer Mausklick in den Zeichenbereich (die Maustas-

te dabei nicht festhalten!) die Position des Mittelpunktes festlegt. Wird die Maus dann weiter bewegt, zieht das den Kreis auf – der Radius ergibt sich aus dem Abstand zwischen der Position des ersten Mausklicks und der aktuellen Position des Mauszeigers. Ein weiterer Klick mit der linken Maustaste legt diesen Radius dann fest und beendet den Zeichenvorgang – der Kreis ist jetzt das erste Element in diesem Beispielprojekt.

Dieser Vorgang hat verschiedene Änderungen innerhalb der Oberfläche von *BeamConstruct* bewirkt: so wurde das neue Element „*Circle*“ rechts in eine Liste eingetragen, welche eine Übersicht über die vorhandenen Elemente eines Projektes bietet. Auf der linken Fensterseite wurde ein Panel „*Circle*“ zu den dort bereits verfügbaren Tabs hinzugefügt. Dieses Panel existiert immer nur so lange, wie ein Element innerhalb des Projektes ausgewählt ist. In diesem lassen sich die Eigenschaften eines solchen Elementes verändern. Im Falle eines Kreises sind das beispielsweise der Linientyp (durchgängig, gestückelt, gestrichelt, Punktmuster, . . .), dessen Anfangs- und Endwinkel sowie einige Parameter mehr, welche zum Teil nur für 3-D-Lasersysteme relevant sind, weil sie den Kreis um die dritte Dimension erweitern.

Eine wirklich interessante Änderung ergibt sich, wenn jetzt in der Toolbar der Button „*Hatch*“ (symbolisiert durch mehrere horizontale Linien in violetter Farbe) betätigt wird. Dieser fügt dem Kreis ein Füllmuster hinzu. Bereits bekannt: das Panel auf der linken Bildschirmseite zeigt jetzt die zu diesem Füllmuster gehörenden Parameter an; hier kann man beispielsweise den Winkel, die Art der Schraffierung, den Abstand von der umrahmenden Geometrie und anderes mehr einstellen. Neu ist die Art, wie dieses neue Element in der Liste auf der rechten Seite angezeigt wird: Jetzt offenbart sich diese als Baumdarstellung, innerhalb derer die Elemente eines Projektes hierarchisch angeordnet sind. So sind Hatch-Patterns



Die Reihenfolge macht's: Gleiche Zusatzelemente mit unterschiedlicher Wirkung, hier im Beispiel eines gehatchten Rechteckes.

(also die Füllmuster) immer einem grafischen Basiselement untergeordnet, da sie ja nur auf bereits vorhandene Geometrien aufbauen können.

Dieses Funktionsprinzip geht aber noch weiter, da es neben dem Hatch auch Elemente zur Nachbearbeitung von Geometrien gibt (in der Toolbar in roter Farbe dargestellt). Diese verändern die vorhandenen Geometrien entsprechend ihrer Eigenschaften und ihrer Position unterhalb eines solchen Basiselements. Soll heißen: eine Liste von Unterelementen wird immer genau in der vorgegebenen Reihenfolge abgearbeitet.

Das sieht auf den ersten Blick etwas kompliziert aus und kann bei umfangreichen Kombinationen von Elementen auch mal zu Verwirrungen führen, bietet tatsächlich aber deutliche Vorteile. So ist es nicht nötig, jedes grafische Element mit Unmengen von Parametern zu überladen, nur um auch noch den letzten geometrischen Kniff aus ihnen herauszuholen. Vielmehr entsteht mit dieser Methode eine Vielfalt von Möglichkeiten allein dadurch, dass Elemente frei kombinierbar sind.

Ein Beispiel ist ein Text, welcher gebogen werden soll, sodass er beispielsweise kreisförmig um etwas herum gelegt werden kann. Hier ist nichts weiter als ein Text-Basisobjekt (blaues Symbol „*Ab*“ in der Toolbar) sowie das „*Curve Distortion*“ Nachbearbeitungsobjekt (rotes Kreissegment-Symbol in der Toolbar) erforderlich. Das erste erzeugt den Text, das zweite – dem Text-Element als Unterobjekt hinzugefügt – verbiegt diesen Text.

Variationen und Kombinationen

An diesem Beispiel des radialen Textes lässt sich auch leicht das Prinzip der Reihenfolge der Unterelemente demonstrieren. Dazu soll in einem ersten Schritt wieder ein Text-Objekt erzeugt werden. Dieses erwartet nur einen einzelnen Mausklick im Zeichenbereich, um dessen Position festzulegen. Dieses Text-Element erhält als erstes Unterelement wieder ein Hatch-Pattern. Hier wird in den Hatch-Parametern lediglich der Linienstil von „*Continuous*“ nach „*Connected Lines*“ geändert. Ist das geschehen, sollte wieder das Textelement selbst selektiert werden, damit im nächsten Schritt das Curve-Nachbearbeitungselement als weiteres Unterobjekt hinzugefügt werden kann.

Jetzt ist zu sehen, dass das Curve-Element alle Geometrien um diesen Radius verbiegt, die bis zu diesem Zeitpunkt existieren, es wird also sowohl der Umriss des Textes als auch sein Füllmuster gebogen. Anders sieht es jedoch aus, wenn anschließend ein weiteres Hatch-Element hinzugefügt wird: Dessen Fülllinien werden durch das Curve-Element nicht beeinflusst, da es in der Reihenfolge der Unterelemente erst nach diesem kommt. Die zweiten Hatch-Linien sind gerade; ungeachtet der zuvor erfolgten Bearbeitung der Geometrien baut dieses Element auf den an dieser Stelle vorhandenen Daten auf.

Dieses Beispiel zeigt deutlich: allein durch die Kombination verschiedener Elemente und der Reihenfolge, in der diese gesetzt werden, sind die unterschiedlichsten Ergebnisse möglich – und das ohne sich mit komplizierten und verwirrenden Parametern herumschlagen zu müssen. Neben dieser logischen Struktur bringen zwar fast alle Elemente noch eigene Parameter mit, diese sind aber recht einfach und übersichtlich, da sie immer nur auf den Kontext des jeweiligen Elementes begrenzt sind und keine „Super-Geometrie“ existiert, die alles können muss.

Die Basiselemente

Diese Elemente – welche in der Applikation selbst als „Primary Element“ bezeichnet werden – werden in der Toolbar am oberen Bildschirm in blauer Farbe dargestellt. Die meisten von ihnen definieren grundlegende Geometrien, einige Sonderfälle erfüllen aber Steuerungsaufgaben und manifestieren sich deswegen auch nicht in Form einer neuen Grafik.

nicht erweiterbare Basiselemente	
Motor	fügt zusätzliche Bewegungsinformationen hinzu, welche bei der Abarbeitung des Projektes den konfigurierten Motor steuert.
Laseroutput	viele Scannercontroller besitzen zusätzliche digitale und analoge Ausgänge; diese können hiermit explizit auf bestimmte Werte gesetzt werden bzw. es ist möglich, Signale in Form von Pulsen auszugeben
Delay	stoppt die Abarbeitung eines Projektes für eine einstellbare Zeitspanne
External Trigger	wartet mit der Abarbeitung aller nach diesem Element folgenden Objekte, bis ein externes Triggersignal an der Scannerkarte festgestellt wurde

Basiselemente	
Dot	ein einzelner Punkt
Line	eine Linie
Circle	ein Kreis (der sich im 3D-Modus auch in Form einer Spiralfeder in die Tiefe fortsetzen kann)
Spiral	eine Spirale (die im 3D-Modus auch in die Tiefe gehen kann)
Rectangle	ein Rechteck
Triangle	ein Dreieck
Star	ein sternförmiges, rotationssymmetrisches Objekt
Polygon	ein frei definierbares, offenes oder geschlossenes Vieleck
Bezier Curve	eine Abwandlung eines Polygons, bei dem die Eckpunkte nicht direkt miteinander verbunden sind sondern die Stützpunkte einer Bezier-Kurve definieren [11]
Text	ein Element zur Textdarstellung; Zeichensatz, Schriftart, Zeichenabstand und vieles mehr sind über die Parameter wählbar
Barcode	erzeugt einen Barcode aus vorgegebenen Daten; Typ, Encoding, Darstellung und anderes mehr sind per Parameter wählbar wobei hier alle wichtigen Barcodetypen wie QR, EAN, DataMatrix sowie diverse Pharma-Barcodes unterstützt werden.

Die Symbole der Basiselemente in der Toolbar.

Basiselemente, welche keine neue Geometrie erzeugen, können logischerweise auch nicht mit Unterelementen um Geometrien erweitert bzw. modifiziert werden.

Das Zusatzelement „Hatch“

Innerhalb der Software existiert zwar ein eigener Elementtyp „Additional Geometry“ (in der Toolbar durch ein violettes Symbol dargestellt), allerdings gibt es derzeit tatsächlich nur ein Element, welches zu dieser Gruppe gehört: der eingangs bereits erwähnte Hatcher. Dieser erlaubt es, bestehende Geometrien zu füllen, um so als Ergebnis einer Laserbearbeitung nicht nur die Kante eines Objektes zu sehen, sondern auch dessen Fläche. Hier kommt der Unterschied zwischen Laserschneiden und -gravieren zum Tragen: würde beim Schneiden die Kante genügen, um ein Objekt mit einem bestimmten Umriss auszuschneiden, so ist beim Lasergravieren die Veränderung der Fläche interessant: Ein beispielsweise auf ein Objekt gelaserter Text soll auch ausgefüllt werden, so dass nicht nur dessen Umriss als Veränderung im Material sichtbar wird, sondern auch die Flächen der Buchstaben.

Das einzige Zusatzelement der Werkzeugleiste ist der Hatcher.

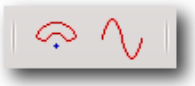
Die Nachbearbeitungselemente

Damit die Wirkung dieser Elemente mit dem offiziellen Namen „Postprocessing Element“ sich auch in sichtbaren und vor allem in den gewollten Änderungen niederschlägt, ist es meistens erforderlich, für die vorangegangenen Basis- oder Zusatzelemente in deren Parametern einen anderen Linientyp als „*Continuous*“ zu wählen. Die Erklärung dafür ist denkbar einfach: Nachbearbeitungselemente können nur bereits existierende Daten verändern, sie fügen keine neuen Geometrien hinzu. Besteht eine lange Linie dann aber nur aus einem Anfangs- und einem Endpunkt, so kann das Element auch nur genau diese beiden Punkte verändern. Und damit lässt sich keine andere Form als eine Linie erzeugen.

Besteht die Kante eines Objektes jedoch aus vielen kurzen Strichen, so können die vielen Stützpunkte dieser Linie auch zu einer neuen Linienform verändert werden.

Sine Distortion – überlagert die Linien einer Geometrie mit einer Sinuswellenform, sowohl die Richtung der Ausbreitung als auch die Amplitude und Frequenz können dabei festgelegt werden.

Curve Distortion – biegt die übergeordneten Geometrien um einen Mittelpunkt herum, dessen relative Position, Ausrichtung, Orientierung und Biegeradius festgelegt werden kann.



Die Nachbearbeitungselemente.

Die Eingabeelemente

Dieser Elementtyp wird in der Toolbar in grün dargestellt und genießt eine Sonderrolle. So kann zu jedem Basiselement maximal ein Eingabeelement existieren. Das aber auch nur dann, wenn das Basiselement mit den Daten eines solchen Eingabeelements umgehen kann.

Die Eingabeelemente heben sich in der Toolbar durch grüne Symbole ab.

Das verdeutlicht sich vielleicht an einem Beispiel: Sowohl der Text als auch das Barcode-Basiselement besitzen in ihrem Konfigurationspanel Eingabefelder, in denen die Daten hinterlegt werden können, welche in Form von Text dargestellt oder in den Barcode hineinverschlüsselt werden können. Genau diese Daten können für diese beiden Elemente aus eben so einem Eingabeelement kommen. Das klingt unspektakulär, ist es im Fall des zweiten Eingabeelements aber definitiv nicht.

OAPC-Input

Wie später noch gezeigt wird, kann ein *BeamConstruct*-Projekt auch innerhalb eines OpenAPC-Visualisierungsprojektes verwendet werden. Dort existiert ein Plug-In, welches mit diesen Projektdaten umgehen kann und welches zusätzliche Dateneingänge besitzt. Genau einer diese Dateneingänge wird über die Parameter dieses Eingabeelements auf das übergeordnete Basiselement umgeleitet. Das heißt nichts

anderes, als dass der Inhalt eines *BeamConstruct*-Projektes auch außerhalb von *BeamConstruct* in einer Maschinensteuerung verändert werden kann, was sich beispielsweise nutzen lässt, um ständig wechselnde Texte auf verschiedene Werkstücke zu bringen

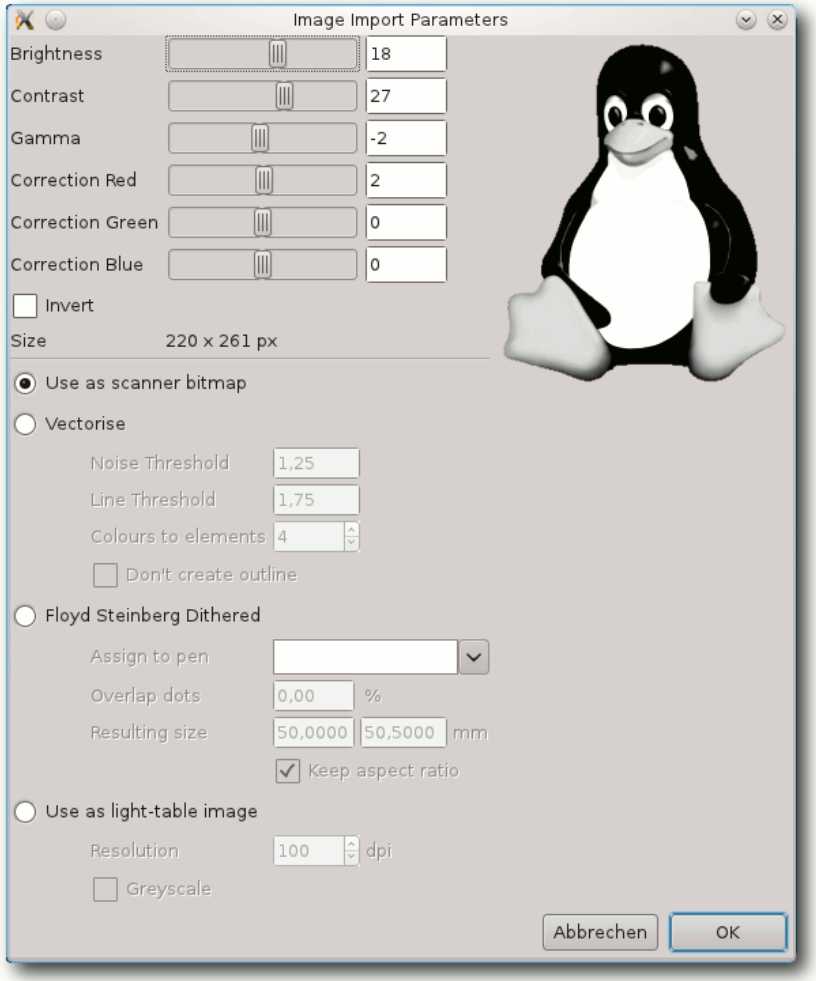
Serial/Date/Time

Der Name dieses Elements deutet bereits an, was damit möglich ist: es lassen sich sowohl Informationen zu Datum und Uhrzeit als auch Seriennummerndaten generieren, welche dann über den Umweg des zugehörigen Basisobjektes auf das zu bearbeitende Material aufgebracht werden können. Dabei sind Kombinationen aus frei wählbarem Text, sich bei jeder Markieroperation verändernden Zählern, Datums- und Zeitinformationen in unterschiedlichster Formatierung möglich, welche es erlauben, die vielfältigsten Anwendungsgebiete abzudecken. So ist es mittels dieses eher unscheinbaren Elementes möglich, so unterschiedliche Dinge wie Produktionsdaten, Schichtinformationen, Verfallsdaten, Seriennummern und vieles andere mehr zu erzeugen – um die Aktualisierung kümmert sich die Applikation dabei vor jedem Markierprozess selbst.

Import von Rastergrafiken

Neben der Möglichkeit, eigene Projekte zu erstellen und Vektorgrafiken diverser Formate zu importieren, können auch sogenannte Pixel- oder Rastergrafiken importiert werden. Das sind nichts anderes als die allseits bekannten Bildformate, welche als JPEG, PNG, GIF, BMP oder anderes mehr daher kommen können.

Abhängig vom verwendeten Lasertyp können damit dann reine schwarz-weiß-Grafiken (z. B. mit CO2-Lasern) als auch echte Graustufen-Bilder (z. B. mit YAG-Lasern [12]) auf ein Werkstück aufgebracht werden.



Der Import-Dialog für Raster-Images, Scanner-Bitmaps können auch anschließend noch weiter verändert werden.

Wurde ein entsprechendes Bild über den Menüpunkt „*Project* → *Import*“ geladen, öffnet sich ein Dialog, in dem grundlegende Bildparameter angepasst werden können. Neben einfachen Möglichkeiten, das Bild noch ein wenig zu verändern, kann hier auch festgelegt werden, wie es innerhalb eines *BeamConstruct*-Projektes eingesetzt werden soll.

Use as Scannerbitmap

Wird diese Option gewählt, so wird das Bild auf eine Art in das Projekt importiert, die dafür sorgt, dass es durch den Laser direkt als echte Bitmap ausgegeben wird. Dabei entscheidet der zuvor für die Scannerkarte gewählte Lasertyp, ob das Bild in echten Graustufen oder als Schwarzweißbild wahlweise mit Floyd-Steinberg-Dithering dargestellt wird

Vectorise

Das Bild wird vektorisiert, d. h. die Pixeldaten werden in Vektorlinien umgewandelt. Dieser Vorgang ist recht kompliziert und das Ergebnis hängt auch sehr stark von den gewählten Vektorisierungsparametern sowie der Qualität des geladenen Bildes ab. Generell gilt: ein starker Kontrast, eine hohe Auflösung, wenig Rauschen und erst recht keine Kompressionsartefakte

sind wichtige Voraussetzungen für brauchbare Ergebnisse. Dennoch sollte man auch dann immer noch eine gewisse Zeit einplanen, um mit den vorhandenen Parametern zu spielen.

Floyd Steinberg dithered

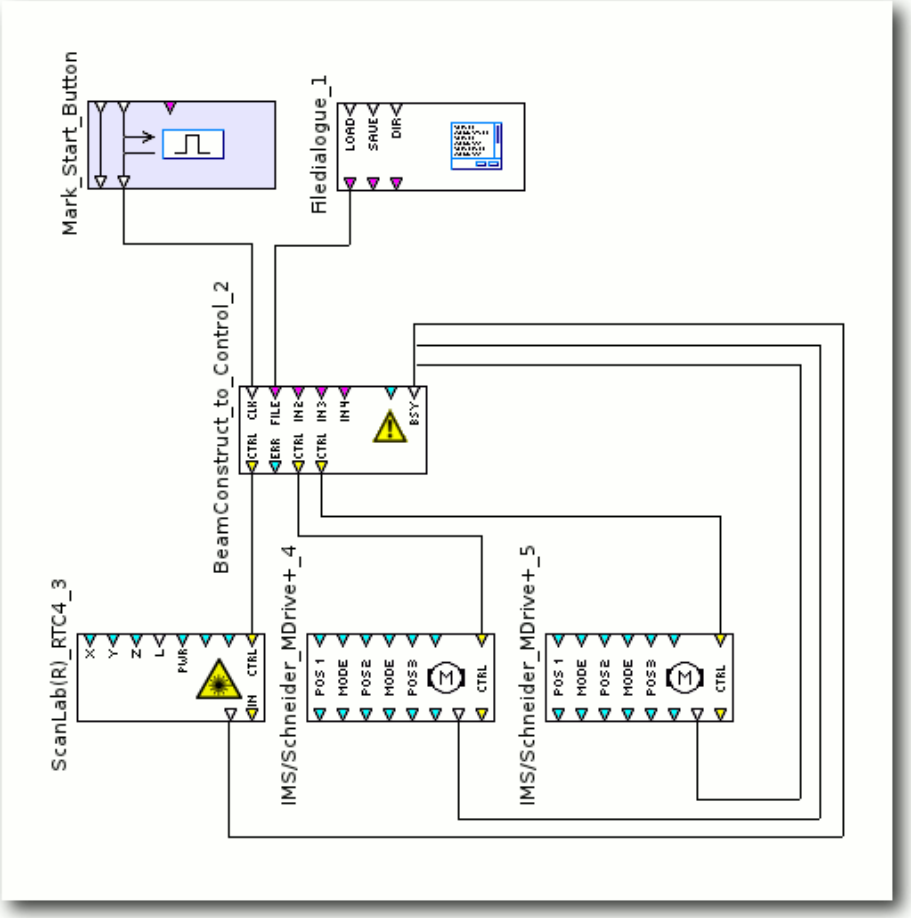
Auch diese Importmethode wandelt das Bild in Vektordaten um – dieses Mal allerdings ausschließlich in einzelne Punkte [13]. Die Anordnung dieser Punkte ist auf eine bestimmte Größe sowie die Spot-Größe des Lasers hin optimiert, sodass nach dem Import keinerlei Skalierfunktionen mehr auf das Ergebnis angewendet werden sollten.

Use as light-table image

Wird diese Option gewählt, so wird das Bild 1:1, ggf. sogar in Farbe und ohne Modifikation in das Projekt übernommen. Allerdings kann so ein Bild nicht über den Laser ausgegeben werden, vielmehr dient es als Vorlage, um eigene Grafiken zu zeichnen. Der Name „Light-table image“ – frei übersetzt also „Leuchttischvorlage“ – weist auch auf diesen Zweck hin; diese Bilder dienen als Vorlage für eigene Geometrien.

Integration in ControlRoom-Projekte

Wurde *BeamConstruct* entsprechend der vorhandenen Hardware konfiguriert, so ist es über den Menüpunkt „*Process* → *Mark*“ möglich, einen Lasermarkierprozess direkt aus der Applikation heraus zu starten. Dabei werden alle vorhandenen Elemente angesteuert: sowohl der Laser als auch der Scankopf arbeiten entsprechend der Vektordaten und den für diese hinterlegten



Mögliche Verdrahtung eines ControlRoom-Projektes mit RTC4 und zwei MDrive-Motoren: alle BSY-Ausgänge müssen mit dem BSY-Eingang des Konverter-Plug-Ins verbunden sein.

Laser- und Scannerparameter (wie beispielsweise Markiergeschwindigkeit, Laserleistung und -frequenz) und die optional vorhandenen Motoren werden so verfahren, wie es im Projekt definiert wurde. Des Weiteren wird vor dem Start einer Laserausgabe ein Projekt auch immer vollständig aktualisiert: so werden dynamische Elemente wie Datum/Zeit oder Seriennummern auf den aktuellen Stand gebracht bzw. weitergezählt.

Da es allerdings die Ausnahme darstellen dürfte, dass die Applikation, mit der die Prozessdaten erstellt werden, auch die zugehörige Maschine ansteuert, gibt es auch noch einen weiteren Weg: Die Prozessvisualisierungs- und -steuerungssoftware „ControlRoom“, welche ebenfalls Teil des OpenAPC-Paketes ist, kann *BeamConstruct*-Projekte laden und abarbeiten. Das geschieht mit Hilfe des zugehörigen Plug-Ins „BeamConstruct to Control“ welches diese Projekte laden und in Control-Daten umwandeln kann. Diese werden dann über den Ausgang OUT0 an den Scannercontroller weiter gegeben. Die Ausgänge OUT2 und OUT3 sind optional, sollten aber mit ein oder zwei Motorcontrollern verbunden sein, sofern die geladenen Projekte diese Motoren erwarten. Existiert so ein Motor jedoch nicht, so wird ein *BeamConstruct*-Projekt, welches diesen

erwartet, bis in alle Ewigkeit auf die Fertigstellung dieser Bewegung warten.

Daraus ergibt sich auch eine weitere Notwendigkeit: Die BSY (=busy)-Ausgänge aller an das *BeamConstruct*-Plug-In angeschlossenen Komponenten müssen mit dessen BSY-Eingang verbunden werden. Über diese Leitung wird signalisiert, ob eine Operation – sei es nun ein Markiervorgang oder eine Motorbewegung – noch läuft oder ob diese fertiggestellt wurde und zum nächsten Bearbeitungsschritt übergegangen werden kann.

BeamConstruct versus CNConstruct

Wie eingangs erwähnt sind sich *BeamConstruct* und CNConstruct sehr ähnlich. Um genauer zu sein, leistet CNConstruct in weiten Teilen das Gleiche wie *BeamConstruct*, allerdings auf einer wesentlich generischeren Basis. Die dort erstellten Geometrien sind nicht bereits für eine bestimmte Art der Bearbeitung optimiert, vielmehr bleibt es hier völlig offen, wie der eigentliche Prozess aussieht.

Anders ist das bei *BeamConstruct*: hier sind verschiedene zusätzliche Funktionalitäten enthalten, welche nur für die Laserbearbeitung sinnvoll sind; auch sind die Penparameter bereits auf das Ausgabemedium „Laser“ abgestimmt. Ein weiterer deutlicher Unterschied: CNConstruct erlaubt nur eine Simulation des Prozesses, *BeamConstruct* kann vorhandene Hardware auch selbst ansteuern und so einen Prozess ablaufen lassen.

Sieht man hiervon einmal ab, sind sich beide aber durchaus ähnlich. So kann man die hier beschriebene Vorgehensweise für das Erstellen von Projekten, Hierarchien und Kombinationsmöglichkeiten von Elementen 1:1 für CNConstruct übernehmen – das grundlegende Bedienkonzept ist bei beiden Applikationen das Gleiche.

Licht versus Schatten

Zu jeder Programmvorstellung gehört immer auch eine gehörige Portion Kritik der Mängel (die jede Software hat), der Softwarefehler und sonstigen Unzulänglichkeiten. So etwas gibt es auch bei *BeamConstruct*; ab und zu findet sich mal ein unerklärlicher Absturz (weswegen es sich empfiehlt, ein Projekt öfter mal zu speichern), hin und wieder tut die Applikation nicht wirklich das, was man erwartet.

Allerdings fällt es schwer, das jetzt als echten Kritikpunkt hervorzuheben, da die hier getestete Version aus dem OpenAPC 2.0 Paket noch als Alpha-Version gekennzeichnet ist. Damit werden eigentlich Softwarestände gekennzeichnet, welche sich in der Regel irgendwo im Zustand „experimentell“ befinden. Das gilt in diesem Fall ganz klar nicht; mit der Software lässt sich arbeiten und sie ist definitiv benutzbar. Es bleibt also abzuwarten, welche Verbesserungen sich bis zur ersten stabilen Release noch ergeben.

Auch wenn aus diesem Grund ebenso mit Lob gespart werden soll, eins bleibt hervorzuheben: dass es mit *BeamConstruct* jetzt auch eine Lasermarkiersoftware für Linux gibt, bedeutet für die gesamte Laserbranche ein echtes Novum – und setzt für die Zukunft hoffentlich Maßstäbe!

LINKS

[1] http://www.freiesmagazin.de/freiesMagazin-2011-01

[2] http://www.freiesmagazin.de/freiesMagazin-2011-03

[3] http://www.freiesmagazin.de/freiesMagazin-2011-06

[4] http://de.wikipedia.org/wiki/Laserscanning#Materialbearbeitung_und_Fertigung

[5] http://de.wikipedia.org/wiki/Laserscanning#Scankopf

[6] http://de.wikipedia.org/wiki/Rapid_Prototyping

[7] http://www.scanlab.de

[8] http://www.raylase.com/

[9] http://www.wxwidgets.org/

[10] http://de.wikipedia.org/wiki/Laser#Laser-Klassen

[11] http://de.wikipedia.org/wiki/Bézierkurve

[12] http://de.wikipedia.org/wiki/Nd:YAG-Laser

[13] http://de.wikipedia.org/wiki/Floyd-Steinberg-Algorithmus

Autoreninformation

Hans Müller ist als Elektroniker beim Thema industrielle Automatisierung mehr der Hardwareimplementierung zugeneigt als dem Softwarepart und hat auch schon diverse Geräte in der privaten Wohnung verkabelt.

Diesen Artikel kommentieren

Google Code-In von Sujeevan Vijayakumaran

Google Code-In [1] ist ein Wettbewerb, der von Google veranstaltet wird und sich ausschließlich an Schüler richtet. Die Teilnehmer tragen bei diesem Wettbewerb zu vielen verschiedenen Open-Source-Projekten bei und können zahlreiche kleinere Aufgaben erledigen. Teilnahmeberechtigt sind alle Jugendlichen zwischen 13 und 18 Jahren, die eine Schule besuchen. Insgesamt beteiligen sich 18 Projekte bei dem Wettbewerb. Darunter sind neben den beiden großen Desktop-Umgebungen GNOME und KDE auch openSUSE und FreeBSD vertreten.

Aufgaben

Insgesamt gibt es für die Teilnehmer acht verschiedene Typen von Aufgaben zu erfüllen. In die Aufgabengruppen fallen nicht nur Programmieraufgaben, sondern auch Dokumentation, Qualitätssicherung und das allgemeine Lösen von Problemen. Des Weiteren können die Teilnehmer an Aufgaben arbeiten, die sich um Marketing und Community-Management drehen. Auch können Hilfestellungen bei der Benutzung eines bestimmten Projektes gegeben oder Problemstellungen zur graphischen Oberfläche bearbeitet sowie Übersetzungen vorgenommen werden.

Jede Aufgabe hat einen bestimmten Status. Bevor man die Arbeit beginnen kann, muss man eine offene Aufgabe finden und diese dann für sich beantragen. So ändert sich der Aufgabenstatus von „open“ zu „claim requested“, bis ein Mentor des Projektes die Aufgabe zur Bearbeitung freigibt. Dann ändert sich der Status zu „claimed“ und der Schüler kann mit der Arbeit beginnen. Für die Bearbeitung wird eine bestimmte Zeit vorgegeben, in der man seine erste Lösung abgeben muss. Der Mentor des dazugehörigen Projekts überprüft die Aufgabenlösung und kann eine weitere Bearbeitungszeit mit Verbesserungsvorschlägen vergeben, sodass Korrekturen an der Lösung vorgenommen werden können. Der Mentor kann die Aufgabe auch wieder zur Bearbeitung freigeben, wenn keine Lösung eingegangen oder etwas völlig Irrelevantes angekommen ist. Bei vollständiger Lösung markiert der Mentor die Aufgabe als vollendet.

Nach der ersten Vervollständigung einer Aufgabe seitens des Teilnehmers muss dieser sich mit vollen persönlichen Daten registrieren, sodass danach die Aufgabe als von ihm erledigt markiert werden kann. Danach kann er eine neue Aufgabe beantragen. Es kann immer nur eine Aufgabe zur selben Zeit beantragt werden.

Jede Aufgabe hat einen Schwierigkeitsgrad, mit dem auch gleichzeitig Punkte vergeben werden. Diese werden in die Grade „leicht“, „mittel“ und „schwer“ einsortiert. Je schwieriger oder aufwendiger die Aufgabe, desto mehr Punkte gibt es. Für schwierige Aufgaben gibt es vier Punkte, für mittlere zwei Punkte, und für leichte einen Punkt. Aus den erzielten Punkten wird dann ein Ranking erstellt.

Preise

Jeder Teilnehmer, der mindestens eine Aufgabe vollständig abgibt, bekommt einen Preis. Sofern er weniger als drei Aufgaben erledigt, bekommt er ein Google Code-In-T-Shirt und eine Urkunde für die erfolgreiche Teilnahme postalisch überreicht. Für jede dritte erledigte Aufgabe winken 100 US-Dollar als Preisgeld. Maximal werden 500 US-Dollar Preisgeld pro Teilnehmer ausgezahlt. Die Anzahl der Aufgaben ist hingegen nicht limitiert, sodass jeder Teilnehmer so viele Aufgaben erledigen kann, wie er möchte. Neben dem T-Shirt und dem Preisgeld werden außerdem zehn „Grand Prize Winners“ gekürt. Diese Gewinner werden anhand des Rankings der erreichten Gesamtpunktzahl ermittelt. Die „Grand Prize Winners“ erhalten eine fünftägige Reise zu Googles Hauptzentrale in Mountain View in den USA und dürfen ein Elternteil als Begleitperson mitbringen. Im letzten Jahr wurden 14 „Grand Prize Winners“ gekürt.

Verlauf

Der Wettbewerb ist in verschiedene Zeitabschnitte unterteilt. Er startete bereits am 21. November 2011. Ab diesem Tag können die Schüler die ersten Aufgaben für die teilnehmenden Projekte beantragen, die die Open-Source-Projekte zuvor zur Bearbeitung freigegeben haben. Die erste Runde endet am 16. Dezember 2011, dann geben die Projekte erneut Aufgaben zur Bearbeitung frei. Google Code-In läuft am 16. Ja-

nuar 2012 aus. Im Februar werden dann die „Grand Prize Winner“ in einem Blogpost genannt. Die Teilnehmer müssen nach Ende des Wettbewerbs ein Dokument von einem Elternteil beziehungsweise Erziehungsberechtigten unterschreiben lassen und in einem Formular hochladen, damit sie die Preise erhalten können.

Fazit

Google Code-In bietet für Schüler durch die vielfältigen Aufgabentypen einen sehr guten Einblick in Open-Source-Projekte. Durch verschiedene Schwierigkeitsgrade und eine breite Anzahl an Aufgaben können sie sich hervorragend einbringen. Gute Englischkenntnisse sind allerdings Voraussetzung, sowohl zum Verständnis der Aufgaben als auch zur Kommunikation mit den Mentoren. Für die deutschsprachigen Teilnehmer gibt es nur wenige Übersetzungsaufgaben zu erledigen, da häufig osteuropäische oder Übersetzungen für andere Sprachen gebraucht werden. Die Programmieraufgaben lassen sich in vielen Teilen nur mit sehr guten Kenntnissen der Sprachen lösen. Vor allem Python und C++ sind gefordert, da sie häufig Anwendung in Open-Source-Projekten finden. Schwierig ist es, diese Aufgaben neben dem Schulalltag zu erledigen, da der Zeitraum zur Erledigung einer Aufgabe teilweise sehr knapp bemessen ist.

Nichtsdestotrotz bietet dieser Wettbewerb einen guten Einblick und Einstieg in Open-Source-Projekte für interessierte Schüler, die mit einem guten Preisgeld auch noch ihr Taschengeld aufbessern können.

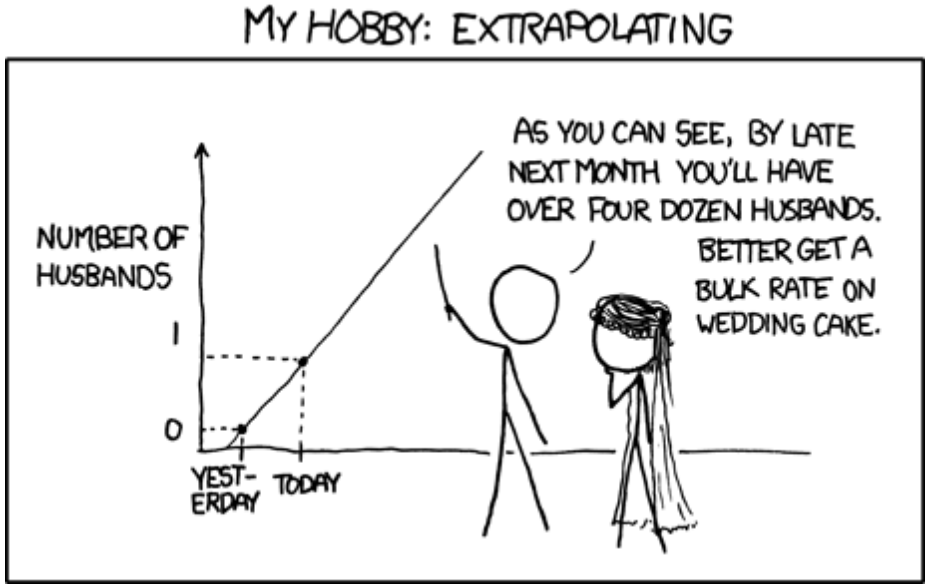
LINKS

[1] http://www.google-melange.com/gci/homepage/google/gci2011

Autoreninformation

Sujeevan Vijayakumaran (Webseite) hat im letzten Jahr am Google Code-In Wettbewerb teilgenommen.

Diesen Artikel kommentieren



„Extrapolating“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/605

Rezension: LibreOffice – kurz & gut

von Michael Niedermair

Das Buch aus der Reihe „kurz & gut“ stellt eine Referenz für das Office-Programm LibreOffice dar, welches aus der Zusammenarbeit von Entwicklern und der Community der kürzlich gegründeten „The Document Foundation“ entstanden ist. Dabei werden die Programme Writer, Calc, Base, Draw und Impress behandelt.

Was steht drin?

Das Buch hat eine kurze Einführung und bespricht dann in sechs Kapiteln die obigen Programme. Abgeschlossen wird die Referenz durch einen Menüindex und ein Stichwortverzeichnis.

Die Bereiche sind im Wesentlichen alle gleich aufgebaut. Dabei werden die jeweils zu dem Programm gehörenden Symbolleisten und Menüs erläutert und die wichtigsten Tastenkürzel dargestellt. Dann erfolgt ein Verweis, auf welcher Seite die Beschreibung zu den Tastenkürzeln folgt.

Der erste Abschnitt (89 Seiten) befasst sich mit der Textverarbeitung Writer. Danach erfolgt die Beschreibung des Formeleditors (10 Seiten). Im dritten Bereich (38 Seiten) geht es um die Tabellenkalkulation mit Calc. Im Anschluss (32 Seiten) folgt das Datenbankfrontend Base. Standardmäßig wird hier HSQLDB verwendet, es kann aber auch jede andere Datenbank (DBMS-System) verwendet werden, wenn ein entsprechender Treiber vorhanden ist. Im fünften Bereich (23 Seiten) geht es um Vektorzeichnungen mit Draw. Der letzte Abschnitt (23 Seiten) beschäftigt sich mit der Präsentationserstellung mit Impress. Zum Schluss folgt ein Menüindex (14 Seiten), der nach Programmen gruppiert die einzelnen Menübefehle/-einträge enthält und auf die entsprechende Seite verweist. Danach kommt das normale Stichwortverzeichnis mit 17 Seiten.

Wie liest es sich?

Das Buch ist eine reine Referenz und zeigt kurz und bündig die notwendigsten Sachen auf, um mit der Office-Suite LibreOffice zu arbeiten. Durch die am Anfang jedes Bereiches dargestellte Funktionsübersicht mit Verweis auf die einzelnen Seiten findet man schnell den entsprechenden Bereich. Passende Screenshots zeigen einem schnell, wo man Einstellungen vornehmen bzw. wie man eine Funktion aufrufen kann.

Kritik

Das Buch ist als Referenz ausgelegt und für diesen Zweck sehr gut geeignet. Durch die Übersicht am Anfang jeden Bereiches, dem Menüindex am Ende und das Stichwortverzeichnis findet man sehr schnell die gesuchte Information. Für das normale Arbeiten und etwas Erfahrung reicht die Referenz sicher aus. Steigt man in die verschiedenen Bereiche tiefer ein, ist Zusatzliteratur mit ausführlicherer Beschreibung notwendig.

Das Buch umfasst 268 Seiten und besitzt bei einem Preis von 12,90 Euro ein sehr gutes Preis-Leistungs-Verhältnis, was man nicht bei allen „kurz & gut“-Büchern sagen kann. Einziger Kritikpunkt zum Inhalt ist der Bereich zum Formeleditor. Hier hätte ich mir etwas mehr gewünscht, z. B. zu aufwändigeren Formeln.

Bedenken habe ich beim Softcover hinsichtlich der dünnen Rückenbindung, da diese nach einer gewissen Zeit ihre Haltefunktion verlieren könnte und Einzelblätter die Folge wären. Der Index ist gut aufgebaut und die Einträge haben meist nur eine Seitenzahl, was dazu beträgt, dass man schnell die entsprechende Stelle findet.

Buchinformationen	
Titel	LibreOffice – kurz & gut (1. Auflage)
Autor	Karsten Günther
Verlag	O’Reilly, August 2011
Umfang	268 Seiten
ISBN	978-3-86899-118-5
Preis	12,90 Euro

Autoreninformation

Michael Niedermair (Webseite) ist Lehrer an der Münchener IT-Schule und unterrichtet hauptsächlich Programmierung, Datenbanken und IT-Technik. Nebenbei schreibt er viel mit LibreOffice und LaTeX.

Diesen Artikel kommentieren

Rezension: CouchDB

von Jochen Schnelle

CouchDB [1] gehört zu den bekannteren Vertretern der Generation der NoSQL-Datenbanken. Was wohl auch daran liegt, dass sich CouchDB relativ leicht in eigene Projekte integrieren lässt, eine einfache API hat und das einige große Projekte wie z. B. Ubuntu One [2] auf CouchDB als Speicherlösung setzen. Jedenfalls ist die Datenbank populär genug, dass sich der Verlag Galileo Computing dazu entschieden hat, ihr ein eigenes Buch zu widmen.

Redaktioneller Hinweis: *Wir danken dem Verlag Galileo Computing für die Bereitstellung eines Rezensionsexemplares.*

Nun gibt es bereits einige Bücher zu CouchDB, unter anderem die auch schon in freiesMagazin besprochenen „Beginning CouchDB“ oder „CouchDB kurz & gut“ (Rezensionen siehe freiesMagazin 08/2010 [3]). Allerdings zielt das vorliegende Buch in eine etwas andere Richtung, wie der Untertitel andeutet. Dieser lautet nämlich: „Das Praxisbuch für Entwickler und Administratoren“. Geschrieben ist das Buch von Andreas Wenk und Till Klamäckel, beide Softwareentwickler, welche nach eigenen Angaben CouchDB schon länger selbst produktiv nutzen.

Inhaltliches

Das rund 300-seitige Buch ist in insgesamt sechs Kapitel unterteilt. Das erste Kapitel namens „Einführung“ widmet sich der Historie von CouchDB, stellt dessen Entwickler kurz vor und führt allgemein in Datenbanken und speziell NoSQL-Datenbanken ein. Im zweiten Kapitel, welches gleichzeitig mit 85 Seiten das längste ist, werden alle Grundlagen und Funktionen von CouchDB erklärt. Dies beginnt mit dem einfachen Anlegen von Datenbanken und Dokumenten, dem Erstellen von Views, List- und Show-Funktionen und behandelt auch etwas fortgeschrittenere Themen wie Replikation, Authentifizierung und URL-Rewriting. Im dritten und vierten Kapitel wird eine Applikation, genauer gesagt ein einfaches Kassenbuch, entwickelt, welche alleine mit CouchDB und Javascript realisiert wird. Der Unterschied zwischen den Kapiteln ist der, dass Kapitel drei auf „nacktes“ Javascript und HTML setzt, während in Kapitel vier CouchApp [4] zum Einsatz kommt, was stark auf jQuery und weitere Javascript-Bibliotheken setzt. Im fünften Kapitel wird die Installation und detaillierte Konfiguration erläutert. Ebenso sind Hinweise und Tipps zur Skalierung von CouchDB zu finden. Das sechste und letzte Kapitel beschreibt die Anbindung von CouchDB an verschiedene Programmiersprachen wie PHP, Ruby, Python und Javascript. Den Abschluss des Kapitels bildet ein kurzer Abschnitt zu Ubuntu One und dessen Nutzung der Datenbank.

Qualitatives

Beim Lesen des Buchs wird schnell klar: die Autoren haben ein sehr umfassendes Wissen, was sie auf jeder Seite an den Leser bringen wollen. Dieses beschränkt sich dabei nicht nur allein auf CouchDB. So werden im ersten Kapitel auch einige Grundlage von Datenbanken wie das ACID-Prinzip, der B-Tree-Index oder das CAP-Theorem behandelt. Alles ist aber praxisnah und kompakt gehalten, es gibt keine Ausschweifungen in die Theorie. Die Praxisnähe ist auch in den anderen Kapiteln wiederzufinden. Immer wieder gibt es Tipps und Tricks für den Alltagsgebrauch und es wird auch vor möglichen Stolpersteinen und Unzulänglichkeiten gewarnt.

Gut ist ebenfalls, dass das Buch auch die zur Zeit aktuellste Version von CouchDB behandelt, nämlich 1.1. An diversen Stellen im Buch finden sich weiterhin Hinweise, seit welcher Version eine bestimmte Funktion zur Verfügung steht, so dass auch Anwender älterer Versionen der Datenbank auf ihre Kosten kommen.

Strukturelles

Bei der Vielzahl der Informationen ist es naturgemäß nicht einfach, diese richtig und in der richtigen Reihenfolge dem Leser näher zu bringen. Das Buch wirkt dadurch an einigen Stellen etwas unstrukturiert, da auf Themen vorgegriffen wird, die erst später abgehandelt werden. Besonders auffällig ist dies z. B. in Kapitel 2. Im Abschnitt 2.2 wird auf die grafische Oberfläche von CouchDB namens Futon eingegangen. Hier wird unter anderem auch erklärt, wie man Views via Futon anlegt. Nur was ein View überhaupt ist, wird

Buchinformationen	
Titel	CouchDB
Autor	Andreas Wenk und Till Klamäckel
Verlag	Galileo Computing
Umfang	304 Seiten
ISBN	978-3-8362-1670-8
Preis	34,90 €

erst später im Abschnitt 2.5. erklärt. Das Buch wird dadurch zwar an keiner Stelle unverständlich oder konfus, nur wer noch nie mit CouchDB gearbeitet hat, der kann stellenweise Schwierigkeiten haben, dem Inhalt zu folgen. Wer hingegen ein wenig Wissen rund um die Datenbank hat, der sollte keine Probleme bekommen.

Fazit

Der Untertitel des Buchs ist Programm: „Das Praxisbuch für Entwickler und Administratoren“. Aktive Anwender – und solche die es werden wollen – finden in dem Buch Unmengen von Informationen, Tipps und Tricks zu allen Bereichen von CouchDB. Nach dem Lesen des Buchs sollten hier kaum noch Fragen offen sein. Für Neueinsteiger ist die Informationsdichte und -geschwindigkeit allerdings eher zu hoch. Hier ist das eingangs erwähnte (englischsprachige) Buch „Beginning CouchDB“ wohl besser geeignet.

Wer sich näher mit CouchDB beschäftigen oder die Datenbank selber aktiv einsetzen möchte, dem kann das Buch aber durchaus empfohlen werden.

Quizfrage

Und weil es ja schade wäre, wenn das Buch bei Jochen Schnelle im Bücherregal verstaubt, verlosen wir „CouchDB“ an die erste Person, die uns folgende Frage beantworten kann:

„CouchDB wird über HTTP-Methoden wie GET, PUT etc. gemäß RFC 2616 angesprochen. Zusätzlich zu den dort definierten Methoden implementiert die Datenbank noch eine weitere Methode außerhalb des Standards. Wie heißt diese?“

Antworten können wie immer über den Kommentarlink am Ende des Artikels oder per E-Mail an redaktion@freiesMagazin.de eingesendet werden.

Links

[1] http://couchdb.apache.org/

[2] http://one.ubuntu.com

[3] http://www.freiesmagazin.de/freiesMagazin-2010-08

[4] http://couchapp.org/page/index

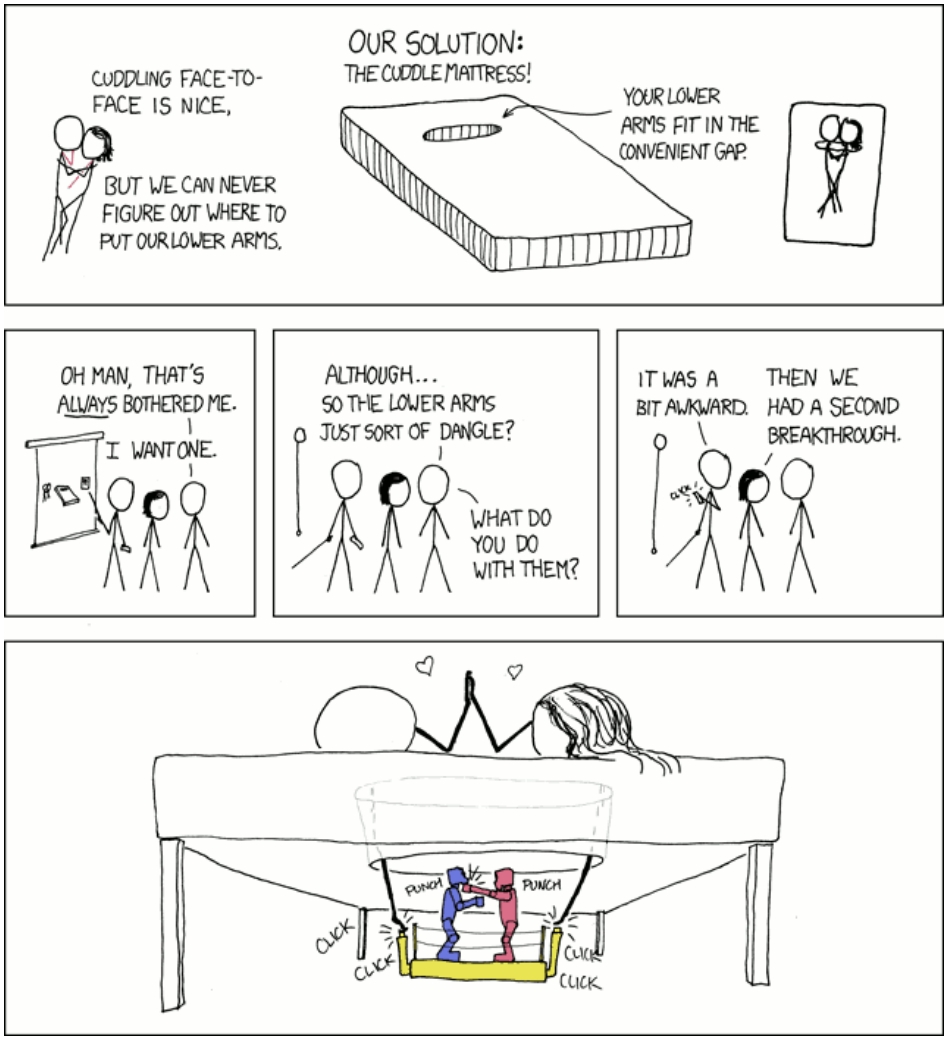
Autoreninformation

Jochen Schnelle (Webseite) nutzt selber CouchDB seit ca. 1,5 Jahren für verschiedene Projekte. Im Buch hat er trotzdem noch einige interessante Informationen gefunden.

Diesen Artikel kommentieren

„Mac/PC“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/934

freiesMagazin-Index 2011



„Mattress“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/335

Vorschau

freiesMagazin erscheint immer am ersten Sonntag eines Monats. Die Januar-Ausgabe wird voraussichtlich am 8. Januar unter anderem mit folgenden Themen veröffentlicht:

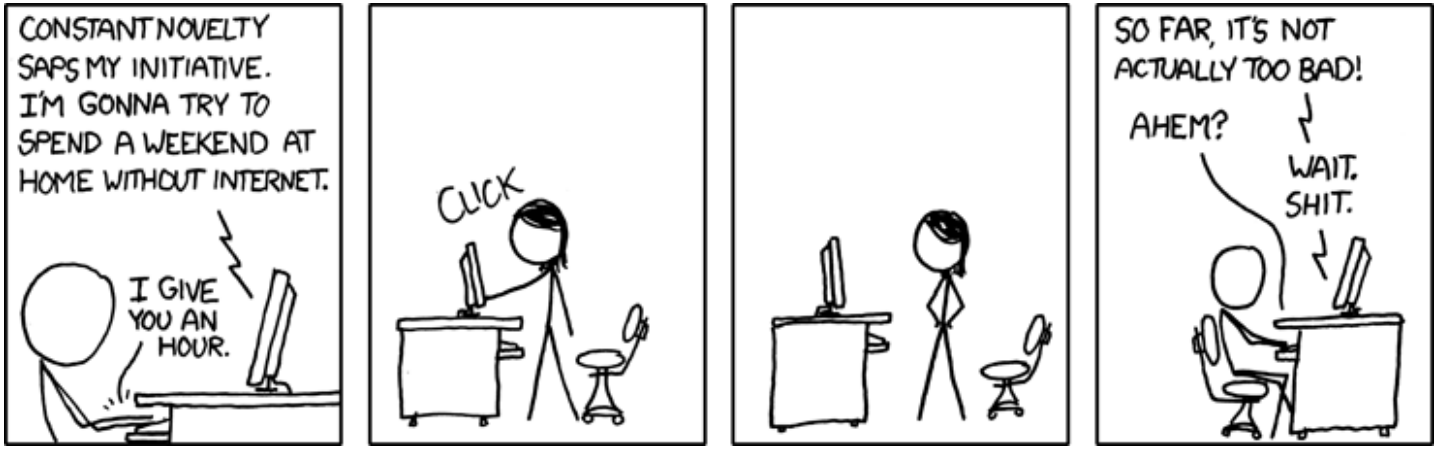
- Fedora 16
- Theme Hospital & CorsixTH
- Rezension: X-Plane kompakt

Es kann leider vorkommen, dass wir aus internen Gründen angekündigte Artikel verschieben müssen. Wir bitten dafür um Verständnis.

Konventionen

An einigen Stellen benutzen wir Sonderzeichen mit einer bestimmten Bedeutung. Diese sind hier zusammengefasst:

$:	Shell-Prompt
#:	Prompt einer Root-Shell – Ubuntu-Nutzer können hier auch einfach in einer normalen Shell ein **sudo** vor die Befehle setzen.
↷:	Kennzeichnet einen aus satztechnischen Gründen eingefügten Zeilenumbruch, der nicht eingegeben werden soll.
~:	Abkürzung für das eigene Benutzerverzeichnis **/home/BENUTZERNAME**
🇬🇧:	Kennzeichnet einen Link, der auf eine englischsprachige Seite führt.
🔍:	Öffnet eine höher aufgelöste Version der Abbildung in einem Browserfenster.



„Addiction“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/597

Impressum

freiesMagazin erscheint als PDF und HTML einmal monatlich.

Kontakt

E-Mail redaktion@**freies**Magazin.de
Postanschrift **freies**Magazin
c/o Dominik Wagenführ
Beethovenstr. 9/1
71277 Rutesheim
Webpräsenz http://www.freiesmagazin.de/

Autoren dieser Ausgabe

Hans-Joachim Baader S. 3
Herbert Breunung S. 34
Patrick Eigensatz S. 31
Tobias Hansen S. 27
Mathias Menzer S. 9
Hans Müller S. 50
Michael Niedermair S. 60
Daniel Nögel S. 40
Urs Pfister S. 44
Jochen Schnelle S. 11, S. 61
Stephan Scholz S. 7
Sujeevan Vijayakumaran S. 58

ISSN 1867-7991
Erscheinungsdatum: 4. Dezember 2011

Redaktion

Dominik Honnef
Thorsten Schmidt
Matthias Sitte
Dominik Wagenführ (Verantwortlicher Redakteur)

Satz und Layout

Ralf Damaschke
Andrej Giesbrecht
Tobias Kempfer
Nico Maikowski
Ralph Pavenstädt

Korrektur

Andreas Breitbach
Daniel Braun
Frank Brungräber
Bastian Bührig
Vicki Ebeling
Stefan Fangmeier
Mathias Menzer
Florian Rummler
Karsten Schuldt
Stephan Walter
Toni Zimmer

Veranstaltungen

Ronny Fischer

Logo-Design

Arne Weinberg (GNU FDL)

Dieses Magazin wurde mit LaTeX erstellt. Mit vollem Namen gekennzeichnete Beiträge geben nicht notwendigerweise die Meinung der Redaktion wieder. Wenn Sie **freies**Magazin ausdrucken möchten, dann denken Sie bitte an die Umwelt und drucken Sie nur im Notfall. Die Bäume werden es Ihnen danken. ;-)