

## Master-Thesis

Name: Adrian Moennich

Thema: Redesign eines webbasierten Conference Management Systems

Arbeitsplatz: CERN, Meyrin

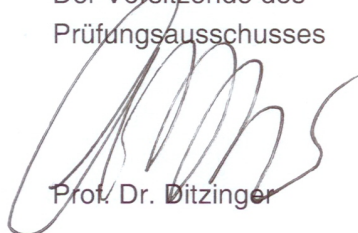
Referent: Prof. Dr. Pape

Korreferent: Prof. Dr.-Ing. Vogelsang

Abgabetermin: 01.10.2013

Karlsruhe, 02.04.2013

Der Vorsitzende des  
Prüfungsausschusses



Prof. Dr. Ditzinger

**Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.**

---

Adrian Mönnich

Karlsruhe, den 27. September 2013

# Zusammenfassung

Im Rahmen dieser Master Thesis soll die am CERN entwickelte *Indico*-Software um ein frei verfügbares Python-Webframework erweitert werden. Bei Indico handelt es sich um eine Webapplikation zur Planung und Verwaltung von Meetings, Konferenzen und ähnlichen Events, wobei auch die Verwaltung und Reservierung von Konferenzräumen integriert ist. Der Einsatz eines modernen Webframeworks erlaubt es, Indico effizienter weiterzuentwickeln, als es mit dem über viele Jahre hinweg in-house entwickelten System derzeit der Fall ist.

Zu Beginn werden die genutzten Technologien Python und WSGI vorgestellt. Danach werden sowohl das derzeitige, speziell auf Indico zugeschnittene, Mini-Framework als auch einige andere Frameworks vorgestellt und mithilfe verschiedener Kriterien analysiert. Aufbauend auf dieser Analyse werden die Vor- und Nachteile der Migration zu einem dieser Frameworks untersucht und anhand dieser ein Framework ausgewählt. Aufbauend auf dem gewählten Framework werden dann Teile von Indico migriert bzw. angepasst.

Die durch diese Thesis erarbeitete Lösung soll dabei eine Grundlage für die Nutzung von verbreitetem, gut dokumentiertem *Third Party*-Code und ein wartbares, entwicklerfreundliches System bilden, welches gleichzeitig auch benutzer- und suchmaschinenfreundlicher als die aktuelle Version ist.

Ein weiteres Ziel neben der Modernisierung des Frameworks ist die Entwicklung und Integration einer einfachen *Recommendation Engine* für Kategorien und/oder Events, sodass Benutzer auf den ersten Blick sehen, was sie evtl. ebenfalls interessieren könnte.

# Abstract

The goal of this master thesis is to extend the *Indico* software developed at CERN with a freely available Python web framework. Indico is a web application to plan and manage meetings, conferences and similar events. Besides managing those, it also allows management and reservation of conference rooms. Using a web framework allows more efficient development than possible with the current framework which has been developed in-house over many years.

At first the used technologies Python and WSGI are introduced. Then both the current Indico-specific framework and various other frameworks are introduced and analyzed according to certain criteria. Based on this analysis the advantages and disadvantages of migrating to one of these frameworks are analyzed and a framework is chosen. By using this framework parts of Indico will be migrated or modified.

The solution developed in this thesis is meant to be a basis for using commonly used well-documented *third party* code and a maintainable developer-friendly system, which will additionally be more user- and search-engine-friendly than the current version.

Another goal besides the modernization of the framework is developing and integrating a simple *recommendation engine* for categories and/or events so users can see possibly interesting or relevant things easily.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Zielsetzung . . . . .	1
1.2	Aufbau der Arbeit . . . . .	2
1.3	CERN . . . . .	3
1.4	Indico . . . . .	4
1.4.1	Überblick . . . . .	4
1.4.2	Aufbau . . . . .	4
1.4.3	Architektur . . . . .	5
<b>2</b>	<b>Grundlagen</b>	<b>7</b>
2.1	Python . . . . .	7
2.1.1	Geschichte . . . . .	7
2.1.2	Anwendungsgebiete . . . . .	7
2.1.3	Erweiterbarkeit . . . . .	8
2.1.4	Implementationen . . . . .	9
2.1.5	Programmierparadigmen . . . . .	10
2.1.6	First-Class-Objekte . . . . .	12
2.1.7	Klassen und Metaklassen . . . . .	13
2.1.8	Lesbarkeit . . . . .	14
2.2	WSGI - Web Server Gateway Interface . . . . .	15
2.2.1	Motivation . . . . .	15
2.2.2	Interface . . . . .	17
2.2.3	Verbreitung . . . . .	18
<b>3</b>	<b>Python-Webframeworks</b>	<b>20</b>
3.1	Vergleichskriterien . . . . .	20
3.2	Indico . . . . .	23
3.3	Django . . . . .	30
3.4	Flask . . . . .	36
<b>4</b>	<b>Auswahl eines Frameworks</b>	<b>45</b>
4.1	Migrationspfade . . . . .	45
4.1.1	Weiternutzung des Indico-Frameworks . . . . .	45

4.1.2	Vollständige Migration zu Django . . . . .	46
4.1.3	Erweiterung durch Django . . . . .	47
4.1.4	Erweiterung durch Flask . . . . .	48
4.1.5	Vollständige Migration zu Flask . . . . .	49
4.2	Entscheidung für ein Framework . . . . .	50
<b>5</b>	<b>Migration zu Flask</b>	<b>53</b>
5.1	Vorbereitung . . . . .	53
5.1.1	Installieren von Flask . . . . .	53
5.1.2	Einbinden von Flask . . . . .	54
5.1.3	Interface zu den mod_python-Funktionen . . . . .	56
5.2	Migration . . . . .	57
5.2.1	Inkompatibilitäten beheben . . . . .	57
5.2.2	Trial and Error . . . . .	60
5.2.3	Routing . . . . .	61
5.2.3.1	Generieren der Routingregeln . . . . .	61
5.2.3.2	Generieren von URLs . . . . .	62
5.2.3.3	Clean URLs . . . . .	64
5.2.3.4	Unterstützung der alten URLs . . . . .	65
5.3	Probleme bei der Migration . . . . .	65
5.4	Ausblick . . . . .	66
<b>6</b>	<b>Dashboard-Erweiterung</b>	<b>67</b>
6.1	Aktueller und gewünschter Zustand . . . . .	67
6.2	Kategorien . . . . .	67
6.3	Events . . . . .	68
6.3.1	Crab . . . . .	69
6.3.2	Probleme . . . . .	70
6.3.3	Alternative . . . . .	70
6.4	Aktueller Stand . . . . .	71
<b>7</b>	<b>Fazit</b>	<b>73</b>
<b>A</b>	<b>Anhang</b>	<b>76</b>
A.1	Screenshots von Indico . . . . .	76
A.2	Quellcodes . . . . .	78
A.2.1	AST einer mod_python-Funktion . . . . .	78
A.2.2	Algorithmus für Kategorie-Scores . . . . .	79

# Abbildungsverzeichnis

1	Indico-Architektur . . . . .	6
2	Open-Source-Lizenz-Kompatibilität . . . . .	23
3	Abstract Syntax Tree . . . . .	62

# Listingverzeichnis

1	ctypes-Beispiel . . . . .	9
2	Zuweisung einer Funktion an eine Variable . . . . .	12
3	Funktion als Funktionsparameter . . . . .	12
4	Klasse wird zur Laufzeit erstellt . . . . .	12
5	Anonyme lambda-Funktion . . . . .	13
6	Python-Klassendefinition . . . . .	14
7	Fehlerhafte Einrückung . . . . .	15
8	Python-CGI-Script . . . . .	16
9	Einfache WSGI-Applikation . . . . .	17
10	Laden der Legacy-Python-Dateien . . . . .	24
11	Einfaches URL-Routing . . . . .	24
12	Mako-Template . . . . .	27
13	Django-Template . . . . .	32
14	Auszug aus der requirements.txt von Indico . . . . .	54
15	Indico-WSGI-Dispatcher . . . . .	55
16	indico.wsgi . . . . .	56
17	mod_python-Wrapper-Factory . . . . .	57
18	about.py im httdocs-Verzeichnis . . . . .	61
19	URLHandler für about.py . . . . .	63
20	Endpointbasierter URLHandler für about.py . . . . .	64



# 1. Einleitung

Ein Webframework ist in einer modernen Webanwendung schon fast Pflicht. Während vor einigen Jahren noch Technologien wie *CGI*<sup>1</sup> oder die Nutzung von PHP in seiner einfachsten Form, d.h. eine Datei pro Seite, die sowohl HTML als auch Geschäftslogik enthält, weit verbreitet waren, so setzen inzwischen immer mehr Entwickler, sowohl von komplexen Webanwendungen als auch von einfachen Websites mit wenigen dynamischen Elementen, auf ein Framework. Dabei stellt sich natürlich die Frage, welche Vorteile ein Framework bietet - schließlich macht es keinen Sinn, ein Framework zu nutzen, nur weil Frameworks „in“ sind.

Ein gutes Framework ermöglicht es dem Entwickler, sich vollständig auf die Anwendungslogik zu konzentrieren, während das Framework alle Aufgaben übernimmt, die in der Regel für jede Anwendung identisch sind, wenn man von Konfigurationseinstellungen absieht. Beispiele für solche Aufgaben sind Sessions, Templates oder der Aufbau einer Datenbankverbindung - keine dieser Bereiche benötigt normalerweise anwendungsspezifischen Code sondern ausschließlich Konfigurationsdaten wie z.B. die Logindaten für die Datenbank oder die Lebensdauer einer Session.

Ein weiterer Vorteil der meisten Frameworks ist die Entkopplung von URLs und Dateisystem. Während in den zuvor erwähnten CGI- oder PHP-Lösungen der Pfad in der URL so gut wie immer einem Pfad im Dateisystem entspricht, erlauben Frameworks normalerweise ein Mapping von URLs (einschließlich dynamischer Elemente) zu Funktionen oder Klassen, sodass der Code der Anwendung unabhängig von der Struktur der URLs sauber strukturiert werden kann.

## 1.1. Zielsetzung

Die *Indico*-Software nutzt WSGI<sup>2</sup> zur Kommunikation mit dem Webserver. Da jedoch ursprünglich das inzwischen veraltete *mod\_python* eingesetzt wurde, und der für die Implementierung von WSGI zuständige Entwickler darauf aufbauenden Code nicht verändern wollte, enthält Indico noch immer eine Kompatibilitätsschicht, die

---

<sup>1</sup>Common Gateway Interface - RFC 3875[RC04]

<sup>2</sup>Web Server Gateway Interface [Eby03]

die von *mod\_python* bereitgestellten APIs emuliert. Diese APIs sind weder entwicklerfreundlich noch unterstützen sie performancerelevante Funktionen wie das Übergeben eines Dateideskriptors an den Webserver, ohne den kompletten Dateiinhalt in den Speicher einzulesen. Bei der Migration zu einem modernen Framework sollte diese Kompatibilitätsschicht vollständig entfernt werden und der Code, der sie nutzt, einem Refactoring unterzogen werden.

Vor der Auswahl eines neuen Frameworks muss zuerst analysiert werden, welche Funktionalität das bestehende Framework zur Verfügung stellt und welche Probleme existieren, damit beim neuen Framework darauf geachtet werden kann, dass diese Probleme dort nicht ebenfalls vorhanden sind. Falls das Framework eine eigene Datenbankschicht enthält muss geprüft werden, wie weit diese mit der in Indico genutzten Objektdatenbank kompatibel ist.

Nachdem ein Framework ausgewählt wurde, muss es zunächst in Indico integriert werden. Danach muss die Anwendung wieder in einen vollständig lauffähigen Zustand gebracht werden, um eventuelle Konflikte zwischen dem Framework und bestehendem Code zu beheben. Ab diesem Zeitpunkt kann analysiert werden, welche Elemente des alten Frameworks neben dem WSGI-Kern ebenfalls durch das neue Framework ersetzt werden können bzw. ob es Teile der Anwendung gibt, in denen es sinnvoll ist, bereits vorhandenen Code zu nutzen statt sie durch eine entsprechende Lösung des neuen Frameworks zu ersetzen.

## 1.2. Aufbau der Arbeit

Im Einleitungskapitel wird auf die Aufgabenstellung eingegangen und die groben Schritte zum Ziel beschrieben. Dann wird kurz auf die Firma eingegangen, an der das Projekt durchgeführt wurde. Ebenfalls wird die Indico-Software, die im Rahmen dieser Arbeit um ein Framework erweitert wird, vorgestellt, um einen Überblick über ihre Funktion zu geben.

Im Grundlagenkapitel werden die genutzten Technologien und ihre Besonderheiten vorgestellt. Nach einem Überblick über die Programmiersprache Python werden ihre Besonderheiten kurz vorgestellt und das WSGI-Interface näher betrachtet.

Auf die Funktionen des bestehenden Frameworks wird im dritten Kapitel eingegangen. Desweiteren werden dort die in Frage kommenden Frameworks vorgestellt und

anhand verschiedener Kriterien analysiert, sodass die Entscheidung für ein Framework später gut begründet werden kann.

Im vierten Kapitel wird die Entscheidung für das Flask-Microframework begründet, nachdem die verschiedenen Migrationspfade erläutert und die jeweiligen Vor- und Nachteile diskutiert wurden.

Die wichtigsten Schritte der eigentlichen Migration werden im fünften Kapitel näher beschrieben. Zum Schluss wird auf die dabei aufgetretenen Probleme eingegangen und ein Ausblick auf mögliche Erweiterungen gegeben.

Zusätzlich zur Frameworkmigration soll das Benutzerdashboard durch automatische Vorschläge zu Kategorien und Events erweitert werden. Lösungsansätze zu diesem Teilprojekt werden im sechsten Kapitel vorgestellt.

Kapitel sieben schließt diese Arbeit mit einem Rückblick ab.

### 1.3. CERN

Beim CERN<sup>3</sup> handelt es sich um eine internationale Forschungseinrichtung im Schweizer Kanton Genf. Der Hauptaufgabenbereich liegt in der Erforschung von Grundlagen der Teilchenphysik, wobei der weltgrößte Teilchenbeschleuniger *LHC*<sup>4</sup> zum Einsatz kommt.

Neben der physikalischen Forschung wird am CERN auch Software entwickelt, die zwar in erster Linie zur internen Nutzung dient, jedoch oftmals auch in Form von *Open Source* der Allgemeinheit kostenfrei zur Verfügung gestellt wird. Diese Software erstreckt sich über viele Bereiche der IT, so werden am CERN z.B. Business-Software für den Human-Resources-Bereich, Grid-Lösungen für die verteilte Datenverarbeitung, Steuersysteme für den Teilchenbeschleuniger und Webanwendungen für die Archivierung von Medien entwickelt.

Die Abteilung *IT-CIS-AVC*<sup>5</sup> ist zuständig für die Verwaltung und Einrichtung von Videokonferenzsystemen, Aufzeichnung und Onlinestreaming von Vorträgen, Meetings und Konferenzen und die Software zu deren Planung. Ebenfalls im Zuständigkeitsbereich der Abteilung sind die Informationsbildschirme, die in verschiedenen

---

<sup>3</sup>Conseil Européen pour la Recherche Nucléaire - Europäische Organisation für Kernforschung

<sup>4</sup>Large Hadron Collider, der Teilchenbeschleuniger am CERN

<sup>5</sup>Collaboration & Information Services - AudioVisual and Collaborative Services

Gebäuden des CERN installiert sind und aktuelle Informationen zum nächstgelegenen Konferenzraum oder aber den Status des *Large Hadron Colliders* anzeigen.

## 1.4. Indico

### 1.4.1. Überblick

Indico<sup>6</sup> ist eine von der Abteilung *IT-CIS-AVC* am CERN entwickelte Webapplikation, die zum Planen und Organisieren von Events dient. Dabei kann es sich sowohl um einfache Vorträge handeln, aber auch um Meetings und mehrtägige Konferenzen mit mehreren Sessions und Beiträgen. Zur Integration in eine bestehende Benutzerinfrastruktur unterstützt die Software außerdem externe Benutzerauthentifizierung. Insbesondere für Meetings und Konferenzen wichtig sind Features wie das *Paper Reviewing*<sup>7</sup>, elektronische Sitzungsprotokolle und ein Archiv für die in einer Konferenz benutzten Materialien. [Ind13]

Diese Featureliste ist jedoch schon lange nicht mehr aktuell, da im Laufe der Zeit immer mehr Funktionen hinzugefügt wurden. Inzwischen enthält Indico u.a. ein Reservierungssystem für Konferenzzimmer, sodass beim Erstellen eines Events sichergestellt werden kann, dass der gewünschte Raum auch verfügbar ist und nicht gerade anderweitig benutzt wird. Diverse Videokonferenzsysteme sind ebenfalls in Indico integriert, sodass diese - sofern sie im jeweiligen Raum verfügbar sind - vollautomatisch eingerichtet und aktiviert werden können. Ein neueres Feature, welches gerade in Entwicklung ist, ermöglicht es den Organisatoren einer Konferenz, Tickets mit QR-Code zu generieren, die vor Ort dann mithilfe einer Smartphone-App eingescannt und auf Gültigkeit geprüft werden können.

### 1.4.2. Aufbau

Kategorien und Events in Indico sind in einer Baumstruktur organisiert: Auf der Rootebene finden sich in der Regel ausschließlich Kategorien, die jeweils entweder weitere Kategorien oder beliebige Events (Vorträge, Meetings und Konferenzen) enthalten können. Ein solches Event wiederum kann diverse Elemente enthalten, wobei

---

<sup>6</sup>Integrated Digital Conference - <http://indico-software.org/>

<sup>7</sup>Im Rahmen eines *Call for Papers* oder *Call for Abstracts* bei einer Konferenz

deren Zusammensetzung vom Typ des Events abhängt. Ein Vortrag oder Meeting enthält in Indico z.B. grundsätzlich kein Registrierungsformular.

Die folgende Auflistung beinhaltet die wichtigsten Bestandteile von Events:

- *Call for Abstracts*
- Chaträume
- Evaluation
- Materialien
- *Paper Reviewing*
- Registrierungsformular (inkl. Integration von Payment-Gateways)
- Videokonferenzen
- Zeitplan

### 1.4.3. Architektur

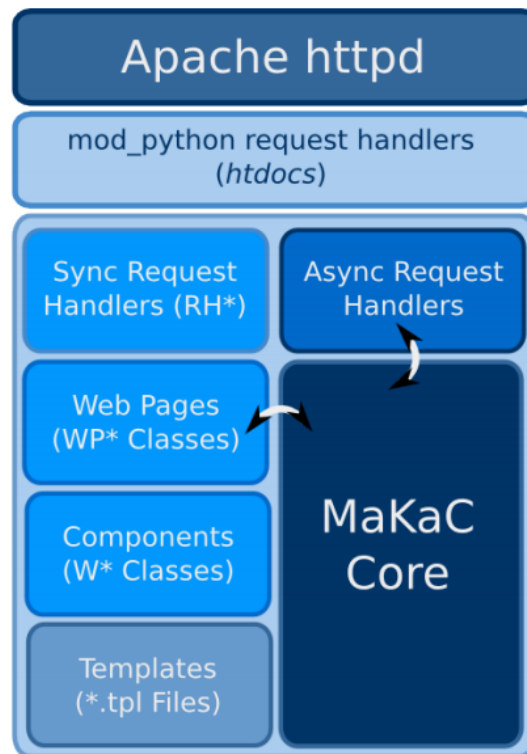
Der serverseitige Code von Indico ist komplett in Python 2.6 geschrieben und nutzt die ZODB<sup>8</sup> als Datenbank. Der Code ist in einer mehrschichtigen Architektur organisiert: Die Anfrage des Benutzers wird vom Webserver über WSGI<sup>9</sup> an die Anwendung weitergegeben, in der die *Request Handler (RH)*-Ebene die Anwendungslogik ausführt. Diese baut die Datenbankverbindung auf, prüft die Zugangsberechtigung des Benutzers, verarbeitet vom Client übermittelte Daten und führt letztendlich auch die gewünschte Aktion aus. Um eine dynamische HTML-Seite auszugeben, nutzt der *Request Handler* die *Web Pages (WP)*-Ebene. Dort werden diverse Aktionen ausgeführt, die die Anzeige der Daten beeinflussen - insbesondere werden dort die benötigten JavaScript- bzw. CSS-Packages mithilfe der *webassets*-Bibliothek geladen. Die eigentliche Erzeugung der HTML-Datei übernimmt die *Components (W)*-Ebene. Bei dieser Ebene handelt es sich um ein Überbleibsel aus der Anfangszeit von Indico, als Templates noch keinerlei Logik enthalten konnten. Jede Klasse in dieser Ebene entspricht einem Template. In der Regel werden dort nur die aus der WP-Ebene übergebenen Daten an das Template weitergereicht.

---

<sup>8</sup>Zope Object Database, <http://www.zodb.org>

<sup>9</sup>Web Server Gateway Interface

Abbildung 1 zeigt die ursprüngliche Struktur, die sich jedoch im Laufe der Zeit verändert hat: Statt des obsoleten *mod\_python*<sup>10</sup> wird inzwischen WSGI verwendet.



**Abbildung 1:** Die Architektur von Indico [Fer08]

---

<sup>10</sup><http://www.modpython.org>

## 2. Grundlagen

### 2.1. Python

#### 2.1.1. Geschichte

Python wurde 1989 von Guido van Rossum am *CWI*<sup>11</sup> entwickelt und 1991 als Betaversion in der Usenet-Newsgroup *alt.sources* veröffentlicht.

Seitdem wurde die Sprache kontinuierlich weiterentwickelt und 1994 in der Version 1.0 veröffentlicht. Die erste Version von Python 2.0 erschien im Herbst 2000 und legte den Grundstein für die heute weit verbreiteten Versionen 2.6 und 2.7, wobei es sich bei 2.7 um die letzte Version von Python 2 handelt.

Version 3 der Programmiersprache ist seit 2008 verfügbar, wobei sie in der Verbreitung bisher noch nicht an Python 2 anknüpfen konnte. Dies liegt hauptsächlich an den teilweise nicht zu Python 2 kompatiblen Änderungen, die zu einem Henne-Ei-Problem führen: Einige populäre Python-Bibliotheken und -Frameworks sind noch nicht mit Python 3 kompatibel, was Entwickler neuer Anwendungen oftmals davon abhält, auf Python 3 zu setzen. Dies führt allerdings wieder zu geringer Nachfrage nach mit Python 3 kompatiblen Bibliotheken. [Wik13]

Um diesem Problem entgegenzuwirken, wurden in den Versionen 2.6 und 3.3 einige Syntaxelemente hinzugefügt, die es Entwicklern einfacher machen, mit beiden Versionen kompatiblen Code zu schreiben. [RC12]

#### 2.1.2. Anwendungsgebiete

Bei Python handelt es sich um eine höhere *general-purpose*-Programmiersprache mit der sich nahezu beliebige Anwendungen einfach realisieren lassen. Es existieren diverse Frameworks sowohl für Webanwendungen als auch für Desktopanwendungen

---

<sup>11</sup>Centrum Wiskunde & Informatica - das nationale niederländische Forschungsinstitut für Mathematik und Informatik

mit grafischer Oberfläche, sodass man als Entwickler in der Regel nur wenig nicht-anwendungsspezifischen Code schreiben muss.

Während es sich bei den Webframeworks in der Regel um reine Python-Frameworks handelt, so sind die GUI-Frameworks meist Bindings für in C oder C++ geschriebene Frameworks wie *Qt*, *GTK+* oder *wxWidgets*, was natürlich den großen Vorteil hat, dass diese Frameworks bereits außerhalb der Python-Community häufig genutzt werden und dementsprechend stabil sind.

Ein weiteres Gebiet, in dem Python häufig verwendet wird, ist das Scripting von diversen Tools. Populäre Editoren wie *Vim* können Python-Skripts zur Automatisierung von Aufgaben nutzen oder auch um beliebige andere Aktionen im Editor auszuführen.

Wie die meisten Open-Source-Programmiersprachen ist Python insbesondere in der Linux-Welt weit verbreitet und wird dort auch oftmals für einfache Systemtools eingesetzt, wo eine Implementierung in C deutlich aufwändiger wäre. So ist der Paketmanager *Portage* der Linux-Distribution *Gentoo* fast vollständig in Python geschrieben; nur einige Bibliotheken sind in C implementiert, da C-Code deutlich performanter ist als interpretierter Code einer Scriptsprache.

### 2.1.3. Erweiterbarkeit

Anders als PHP, wo Erweiterungen ausschließlich in C geschrieben werden können, bietet Python verschiedene Möglichkeiten, mit nicht-Python-Code zu interagieren. Natürlich können native Erweiterungen für Python in C oder einer anderen Sprache, die zu einer *Shared Library* kompilierbar ist, implementiert werden und haben dadurch vollen Zugriff auf alle Lowlevel-APIs von Python. Dies ist allerdings oftmals nicht notwendig und führt nur zu unnötig komplexem Code, da die C-APIs auf einer sehr niedrigen Ebene arbeiten und auch die *Reference Counts* des Garbage Collectors vom Entwickler penibel aktualisiert werden müssen, um Speicherlecks zu vermeiden.

Ein häufiger Grund für native Erweiterungen ist das Zugänglichmachen von existierenden Bibliotheken, die in C geschrieben sind. Auf den ersten Blick würde man vermuten, dass es in diesem Fall sinnvoll ist, eine native Erweiterung zu schreiben. Wenn man bedenkt, was das Ziel eines solchen Interfaces ist, merkt man jedoch schnell, dass meist weder Lowlevel-APIs noch die erhöhte Performance von C-Code notwendig ist - alles was benötigt wird ist eine Möglichkeit, Funktionen aus einer



kompierten Bibliothek aufzurufen. Dazu bietet Python mit dem in der Standardbibliothek enthaltenen *ctypes*-Paket ein einfach nutzbares FFI<sup>12</sup>, mit welchem ein Entwickler in reinem Python-Code beliebige Funktionen aus *Shared Libraries* aufrufen und sogar Python-Funktionen als Callback übergeben kann.

---

```
1 import ctypes
2
3 _libcrypt = ctypes.cdll.LoadLibrary('libcrypt.so')
4
5 crypt = _libcrypt.crypt
6 crypt.restype = ctypes.c_char_p
7 crypt.argtypes = [ctypes.c_char_p, ctypes.c_char_p]
8
9 print crypt('secret', '$6$randomsalt')
10 # $6$randomsalt$IJynOtjaPVh8n1ZrY6d.YwXK98nJZCZ4v3fkC7...
```

---

**Listing 1:** ctypes-Beispiel

Auch wenn die Python-Erweiterung performancekritische Operationen enthält und deshalb nicht in Python implementiert werden soll, gibt es eine Möglichkeit, fast reinen Python-Code dafür zu schreiben. *Cython*<sup>13</sup> ist eine mit C-Elementen erweiterte Abwandlung von Python, die zu nativem Code kompiliert wird. Insbesondere um Python-Klassen zu erstellen ist Cython ideal geeignet, da die Cython-Syntax dabei große Ähnlichkeit mit der Python-Syntax hat und man nicht die unintuitive C-API nutzen muss.

### 2.1.4. Implementationen

Wenn die Rede von „Python“ ist, ist in den meisten Fällen die Referenzimplementation *CPython* gemeint. Dabei handelt es sich um den ersten und auch am weitesten verbreiteten Python-Interpreter der, wie der Name schon erahnen lässt, in C geschrieben ist. Er ist Open Source und wird unter anderem von dem Python-Erfinder Guido van Rossum entwickelt.

---

<sup>12</sup>Foreign Function Interface

<sup>13</sup><http://cython.org>

Neben CPython existierten weitere Python-Implementationen. Eine der bekanntesten ist das Java-basierte *Jython*, welches Python-Code zu Java-Bytecode kompiliert und in einer JVM ausführt. Dadurch lässt sich Python-Code einfach in eine existierende Java-Infrastruktur integrieren und kann auf beliebige Java-Packages zugreifen.

Insbesondere für Windows-Desktopanwendungen interessant ist das von Microsoft als Open Source veröffentlichte *IronPython*. Ähnlich wie Jython läuft auch IronPython in einer sprachunabhängigen Runtime-Umgebung - allerdings nutzt IronPython Microsofts .NET-Framework oder alternativ das plattformunabhängige *Mono* und kann damit unter anderem auf das WinForms-GUI-Framework und andere .NET-APIs zugreifen.

Neben diesen Implementierungen existieren noch weitere, die auf spezielle Anwendungen zugeschnitten sind: *Stackless Python* bietet extrem speicherschonende Pseudo-Threads, während *PyPy* ein in Python geschriebener Python-Interpreter ist, der einen JIT-Maschinencode-Compiler enthält und somit teilweise bessere Performance als CPython erzielen kann. Ein eher unbekannter Python-Interpreter ist *PyMite*, bei dem es sich um eine für 8-Bit-Mikrocontroller optimierte Python-Version handelt, die zwar weder die Python-Standardbibliothek noch den vollen Funktionsumfang von Python unterstützt, aber dafür trotz der Einschränkungen einfacher Mikrocontroller funktioniert.

### 2.1.5. Programmierparadigmen

Das Paradigma einer Programmiersprache ist „die Sichtweise auf und den Umgang mit den zu verarbeiteten Daten und Operationen“. [HV06, Kap. 1.3.1] In Python kann man man sich zwischen drei verschiedenen Paradigmen entscheiden.

#### **Imperative/prozedurale Programmierung**

Bei der imperativen Programmierung wird eine Folge von Anweisungen sequentiell abgearbeitet. Zur Kapselung und Wiederverwendung von Funktionalität werden Funktionen genutzt. [HV06, Kap. 1.3.1]

Abgesehen von diversen Modulen in Pythons Standardbibliothek und Exceptions, die, um eine Hierarchie zu ermöglichen, als Klassen realisiert sind, kann ein Python-Programm komplett imperativ geschrieben werden. Dies hat den großen Vorteil, dass insbesondere kleine Kommandozeilenprogramme oftmals

sehr einfach realisiert werden können, da man nicht von der Programmiersprache dazu gezwungen wird, Klassen einzusetzen ohne dass sie für die Strukturierung der Anwendung nützlich sind.

### Objektorientierte Programmierung

Eine objektorientierte Programmiersprache unterstützt Klassen und Objekte, wobei, anders als bei einer nur objekt**basierenden** Programmiersprache, auch komplexere Konzepte wie Vererbung und Polymorphie genutzt werden können. [HV06, Kap. 1.3.1]

Wie zuvor schon erwähnt, nutzt Python unter anderem in Teilen der Standardbibliothek und für Exceptions Klassen. Dementsprechend ist es nur logisch, dass in Python objektorientiert entwickelt werden kann. Dazu bietet Python dem Entwickler ein ausgereiftes Klassensystem, das neben der in objektorientierten Sprachen üblichen Vererbung auch Mehrfachvererbung und sogenannte *Metaclasses* unterstützt. Da es sich gerade bei letzterem um ein in Python einzigartiges Feature handelt, wird im Verlauf dieses Kapitels näher darauf eingegangen.

### Funktionale Programmierung

Eine rein funktionale Programmiersprache basiert nicht auf Wertzuweisungen sondern benutzt ausschließlich Funktionsdefinitionen, die eine Eingabe in eine Ausgabe transformieren. [HV06, Kap. 1.3.1]

Python ist allerdings nicht direkt eine funktionale Programmiersprache und damit erst recht keine rein funktionale Programmiersprache. Dadurch, dass Funktionen *First-Class-Objekte* sind und sowohl Lambdafunktionen als auch Closures unterstützt werden, kann man in Python sehr einfach funktionale Elemente mit objektorientierten bzw. prozeduralen Elementen mischen. Die prominentesten aus anderen funktionalen Programmiersprachen bekannten Methoden sind `map()`, um eine Liste mittels einer Funktion in eine neue Liste zu transformieren, `filter()`, um nur die Elemente aus einer Liste zu übernehmen, die von der Filter-Funktion durch die Rückgabe von `True` akzeptiert wurden, und diverse Werkzeuge für Funktionen höherer Ordnung wie `lambda` und partielle Funktionsapplikation.

### 2.1.6. First-Class-Objekte

Eines der Designziele bei der Entwicklung von Python war, allen Objekten *First-Class*-Status [vR09] zu geben. Dieses Ziel wurde bei allen Sprachelementen konsequent beachtet, sodass neben Funktionen auch Klassen, Datentypen und Module diesen Status haben und dementsprechend Attribute und damit auch Methoden besitzen können. Details zu den verschiedenen Objekttypen finden sich in der Python-Dokumentation<sup>14</sup>.

*First-Class* beschreibt Eigenschaften die man bei Objekten, d.h. meist Instanzen von Klassen, i.a. erwartet, aber im Zusammenhang mit Funktionen, Klassen und anderen Datentypen durchaus ungewöhnlich sind:

- Sie können in Variablen und anderen Datenstrukturen gespeichert werden.

```
1 def func():  
2     pass  
3 something = func
```

**Listing 2:** Zuweisung einer Funktion an eine Variable

- Sie können Funktionsparameter sein.

```
1 roots = map(math.sqrt, [4, 9, 16])
```

**Listing 3:** Funktion als Funktionsparameter

- Sie können Rückgabewert einer Funktion sein.
- Sie können zur Laufzeit erstellt werden.

```
1 def make_class(cls=object):  
2     class _extended(cls):  
3         foo = 'bar'  
4     return _extended
```

**Listing 4:** Klasse wird zur Laufzeit erstellt

---

<sup>14</sup><http://docs.python.org/2/reference/datamodel.html>

- Sie sind nicht an einen Namen gebunden.

```
1 def make_power(power):  
2     return lambda num: num ** power
```

**Listing 5:** Anonyme lambda-Funktion

### 2.1.7. Klassen und Metaklassen

Die Grundfunktionen des Klassensystems in Python sind mit denen anderer objekt-orientierter Programmiersprachen vergleichbar. Eine Klasse kann von beliebig vielen Klassen erben (Mehrfachvererbung) und Methoden dieser Klassen überschreiben, wobei diese Methoden mittels der `super()`-Funktion weiterhin erreichbar sind.

Die Tatsache, dass in Python keine `private`- oder `protected`-Schlüsselwörter existieren, dürfte gerade für einen Java- oder C++-Entwickler sehr ungewohnt sein. Dies liegt daran, dass in Python das Motto „we’re all consenting adults here“ [Fas03] gilt und dementsprechend `private` Elemente einer Klasse nicht technisch erzwungen sondern alleine anhand des Namens als privat markiert werden. So gilt eine Variable oder Methode, die mit einem einzelnen Unterstrich beginnt (`_foo`), als Teil der internen API der Klasse. Dies bedeutet in der Regel, dass sie nicht Teil der Dokumentation ist und die Nutzung auf eigene Gefahr geschieht - man muss damit rechnen, dass sie in einer beliebigen neuen Version wegfällt oder sich ihr Verhalten anderweitig ändert. Da es aber durchaus auch einen technischen Grund für `private` Variablen gibt - nämlich die Vermeidung von Konflikten in Subklassen - kann durch die Verwendung von zwei Unterstrichen (`__foo`) das sogenannte *name mangling* aktiviert werden. Dies führt dazu, dass bei jedem Zugriff auf die Variable bzw. Methode der Name in `_ClassName__foo` umgeschrieben wird und somit eine Subklasse denselben Namen nutzen kann, ohne Konflikte zu verursachen. Es ist aber jederzeit möglich, direkt auf `_ClassName__foo` zuzugreifen.

Eine ähnliche Namensgebung wird für den Konstruktor `__init__` und die Überladung von Operatoren wie `__add__` für den binären Plus-Operator verwendet. Der Python-Styleguide empfiehlt daher, niemals eigene Namen zu verwenden, die mit zwei Unterstrichen beginnen und enden, sofern sie nicht bereits dokumentiert sind und dementsprechend eine besondere Bedeutung haben.

Eines dieser „magischen“ Attribute ist `__metaclass__`, mit dem die Metaklasse einer Klasse beeinflusst werden kann. Um das Konzept von Metaklassen zu verstehen ist zunächst ein Einblick nötig, wie Python eine Klasse erstellt. Als Beispiel soll dazu die sehr einfache Klasse in Listing 6 dienen. Sie erbt von der internen Klasse `object` und enthält eine einzige Variable `a` mit dem Standardwert 1.

```
1 class Test(object):  
2     a = 1
```

---

**Listing 6:** Python-Klassendefinition

Wie zuvor schon erwähnt sind Klassen in Python First-Class-Objekte und können zur Laufzeit erstellt werden. Dies ist natürlich auch bei der Beispielsklasse möglich, der Code dazu ist `type('Test', (object,), dict(a=1))`. Auch ohne Blick in die Dokumentation ist die Funktionsweise von `type()` offensichtlich: Die Funktion erwartet als Parameter den Namen der Klasse, die Liste der Superklassen und das *dict* der Klasse, d.h. ein Mapping, welches alle Variablen und Methoden der Klasse enthält.

Bekanntermaßen sind die meisten Objekte Instanzen von Klassen. Dies trifft in Python auch für Klassen selbst zu, die ebenfalls Objekte sind. Jede Klasse ist eine Instanz ihrer Metaklasse; im Standardfall ist dies `type`, allerdings kann eine beliebige andere Klasse oder Funktion als `__metaclass__` angegeben werden, solange sie dieselben Parameter wie `type()` akzeptiert. Dies ermöglicht es einem fortgeschrittenen Entwickler, die Definition einer Klasse beliebig zu beeinflussen und somit z.B. eine deklarative Syntax für ein ORM<sup>15</sup> zu ermöglichen oder alle Subklassen einer bestimmten Klasse einer Liste hinzuzufügen.

### 2.1.8. Lesbarkeit

Die Sprache legt großen Wert auf lesbaren Code, was insbesondere daran zu erkennen ist, dass Blöcke nicht wie in den meisten anderen Sprachen durch geschweifte Klammern oder *begin/end*-Statements definiert werden, sondern einzig und allein durch die Einrückung. Dementsprechend führt inkonsistente Einrückung auch zu einer entsprechenden Fehlermeldung, wie man in Listing 7 sehen kann.

---

<sup>15</sup>Object Relational Mapper

Während die erzwungene Einrückung insbesondere bei Python-Anfängern oftmals negativ aufgenommen wird, ist der Sinn dahinter klar zu erkennen: Gerade bei Projekten mit mehreren Entwicklern wird so eine einheitliche Einrückung erzwungen, was die Lesbarkeit des Codes stark erhöht.

Allgemein legt Python großen Wert auf „schönen“ Code, was auch im *Zen of Python*<sup>16</sup> festgeschrieben ist: „Beautiful is better than ugly.“ und „Readability counts.“ [Pet04]

---

```
1 >>> def fail():
2 ...     a = 'b'
3 ...     b = 'c'
4   File "<stdin>", line 3
5     b = 'c'
6     ^
7 IndentationError: unexpected indent
```

---

**Listing 7:** Fehlerhafte Einrückung

## 2.2. WSGI - Web Server Gateway Interface

### 2.2.1. Motivation

Um den Nutzen eines standardisierten, auf die Programmiersprache zugeschnittenen, Interfaces zwischen Webserver und Anwendung deutlich zu machen, ist zunächst ein Blick auf die älteste und einfachste Methode zum Ausführen interaktiver Scripts innerhalb eines Webserver angebracht: Das Common Gateway Interface<sup>17</sup>).

Bei der Nutzung von CGI wird für jeden Aufruf einer entsprechenden URL ein neuer Prozess gestartet; Metadaten wie die IP-Adresse des Clients oder die abgerufene URL werden via Umgebungsvariablen weitergegeben. Die Standarddatenströme *stdin* und *stdout* dienen der Übergabe von Request-Bodies (z.B. bei *HTTP POST*) bzw. dazu, die Antwort des Programms an den Webserver zu übergeben.

---

<sup>16</sup>Kann über das Easter-Egg `import this` in der Python-Shell angezeigt werden

<sup>17</sup>CGI, RFC 3875 [RC04]

CGI ist daher sehr einfach zu implementieren. Python bietet mit dem `cgi`-Modul Hilfsfunktionen an, die die Kommunikation über das Common Gateway Interface abstrahieren und Standardaufgaben wie das Parsen von Formulardaten übernehmen. Allerdings muss für jeden Request ein neuer Prozess gestartet und jeglicher Initialisierungscode erneut ausgeführt werden. Dieser Overhead ist außer bei sehr einfachen Anwendungen mit wenigen Benutzern problematisch und selbst dort würde er zu unangenehm langen Ladezeiten führen, wenn komplexe Systeme wie ein ORM-Framework initialisiert werden müssten.

Listing 8 zeigt ein kleines Beispielscript, welches einen GET-Parameter bzw. einen Standardtext ausgibt. Man erkennt an dem manuell ausgegebenen *Content-type*-Header gut, dass CGI sehr simpel ist und das HTTP-Protokoll nur minimalst abstrahiert.

---

```
1 #!/usr/bin/env python
2
3 import cgi
4
5 form = cgi.FieldStorage()
6 print 'Content-type: text/plain'
7 print
8 print 'Hello %s' % form.getfirst('name', 'World')
```

---

**Listing 8:** Python-CGI-Script

Um die CGI-typischen Probleme zu vermeiden, wurde vor etwa dreizehn Jahren das Modul *mod\_python* für den weit verbreiteten Apache-Webserver veröffentlicht. Es ermöglichte Entwicklern erstmals, Python ähnlich komfortabel wie PHP zu nutzen, aber ohne die Nachteile von CGI in Kauf nehmen zu müssen. Der große Nachteil bei *mod\_python* ist jedoch, dass es ausschließlich für den Apache-Webserver verfügbar ist und auch die zur Verfügung gestellten APIs sehr Apache-spezifisch sind. Dies bedeutet, dass man bei der Nutzung von *mod\_python* die in CGI vorhandene Plattformabhängigkeit verliert und der Umstieg auf einen anderen Webserver nur mit viel Aufwand möglich ist.

Im Gegensatz zum sprachunabhängigen CGI und *mod\_python* bietet Java mit der *Servlet*-API eine saubere und gut dokumentierte Schnittstelle für Webanwendungen, sodass man ein beliebiges Framework und einen beliebigen Webserver nutzen kann,



sofern beide die Servlet-API implementieren. Daher wurde mit dem *Web Server Gateway Interface* ein Python-Standard verabschiedet, der für das Zusammenspiel zwischen Webserver und Webanwendung eine einfach zu implementierende Schnittstelle definiert.

### 2.2.2. Interface

Der grundlegende Aufbau von WSGI lässt sich am Einfachsten anhand eines Beispiels beschreiben. Listing 9 zeigt eine sehr einfache *Hello World*-WSGI-Anwendung, an der man aber dennoch gut erkennen kann, wie das Interface aufgebaut ist.

---

```
1 def application(environ, start_response):
2     start_response('200 OK', [('Content-Type',
3                               'text/plain')])
4     yield 'Hello World!'
```

---

**Listing 9:** Einfache WSGI-Applikation

Eine WSGI-Anwendung enthält ein globales *callable object*, das typischerweise den Namen `application` besitzt. Dieses Objekt kann wie im Beispiel eine einfache Funktion sein, aber auch eine Klasse oder eine Klasseninstanz, die `__call__` implementiert. Letzteres ist dabei die insbesondere in größeren Frameworks bevorzugte Methode, da das Objekt alle frameworkspezifischen Daten wie z.B. eine Routingtabelle für URLs enthalten kann.

Alle für WSGI nötigen Daten werden als Parameter übergeben. `environ` enthält dabei die bereits aus CGI bekannten Umgebungsvariablen wie `REQUEST_METHOD` für die HTTP-Methode (`GET`, `POST`, ...) und `QUERY_STRING` für die GET-Parameter in der URL. Neben diesen Variablen sind weitere, WSGI-spezifische, Variablen enthalten. Diese übermitteln u.a. neben der verwendeten Version der WSGI-Spezifikation auch die Datenstreams für den *Request Body* (in CGI: `stdin`) und Fehlermeldungen (in CGI: `stderr`).

Beim zweiten Parameter, `start_response`, handelt es sich genau wie beim Applikationsobjekt um ein *callable*. Bei einem erfolgreichen Request wird es von der Anwendung bzw. dem Framework genau einmal aufgerufen; als Parameter werden der HTTP-Statuscode (in der Regel `200 OK`) und eine Liste mit den HTTP-Antwortheadern übergeben. Sofern bei der Abarbeitung des Requests ein Fehler

auftritt, kann die Funktion erneut aufgerufen werden, um einen anderen Statuscode (meist *500 Internal Server Error*) und die Details des aufgetretenen Fehlers (insbesondere den Stacktrace) zurückzugeben.

Zu diesem Zeitpunkt stellt sich oftmals die Frage, wieso Statuscode und Header nicht einfach mittels `return` zurückgegeben werden, statt den Umweg über einen separaten Funktionsaufruf zu gehen. Dazu ist ein Blick auf den Aufbau einer HTTP-Antwort notwendig: Statuscode und Header stehen immer am Anfang der Antwort. Danach folgt der durch eine Leerzeile von den Headern abgegrenzte Body, der die eigentliche Antwort enthält - oftmals ein HTML-Dokument. Während es bei einer typischen Webseite kein Problem wäre, diesen Body komplett in einer Variable zu speichern und am Ende zusammen mit den Headern zurückzugeben, so würde es bei größeren Daten unnötig viel Arbeitsspeicher verbrauchen und es könnten erst Daten an den Client gesendet werden, wenn die Antwort vollständig vorhanden ist. Um dieses Problem zu vermeiden, ist der Rückgabewert beim Aufruf des Applikationsobjekts ein *iterable*. Dabei kann es sich einfach um eine Liste handeln, deren einziges Element ein String mit der kompletten Antwort ist, aber alternativ auch um einen „echten“ Generator, der die Daten erst beim Iterieren erzeugt - letzteres bietet sich gerade bei größeren Daten oder Archiven an, die in diesem Fall blockweise gelesen und gesendet bzw. on-the-fly erzeugt werden können. Aufgrund dieser Funktionsweise könnte die Antwort in Listing 9 auch mit `return ['Hello World!']` zurückgegeben werden, statt das für Generatorfunktionen typische `yield` zu benutzen.

Um bestehende Anwendungen, die eine `write(data)`-Funktion erwarten, ebenfalls ohne größeren Aufwand WSGI-kompatibel machen zu können, gibt die Funktion `start_response()` eine solche Funktion zurück. Die WSGI-Spezifikation rät allerdings von der Nutzung dieser Funktion ab, da sie diverse Nachteile mit sich bringt. Anders als bei einer Generatorfunktion kann der Webserver dort nicht jederzeit die Abarbeitung unterbrechen sondern es ist alleine der Anwendung überlassen, wann sie Daten sendet.

### 2.2.3. Verbreitung

Jedes aktuelle Python-Webframework basiert inzwischen auf WSGI, wobei oftmals entsprechende Wrapper für CGI und FastCGI zur Verfügung gestellt werden, von deren Nutzung allerdings in der Regel abgeraten wird, sofern es irgendwie möglich ist, WSGI direkt zu nutzen.

Seitens der Webserver ist WSGI ebenfalls verbreitet, wobei dort verschiedene Ansätze genutzt werden: Der Apache-Webserver integriert mit *mod\_wsgi* einen WSGI-Container direkt im Webserver, was eine sehr einfache Konfiguration über die bereits existierenden Webserver-Konfigurationsdateien zur Folge hat. Nichtsdestotrotz unterstützt *mod\_wsgi* alle wichtigen Features - insbesondere separate Prozesse für den Python-Interpreter. Andere Webserver wie *nginx* setzen auf einen externen WSGI-Container wie *uWSGI* oder sind wie *Gunicorn* direkt in Python geschrieben und unterstützen WSGI nativ. Der große Vorteil eines Containers wie *uWSGI* ist die strikte Trennung zwischen Webserver und Python, sodass interaktiver Code problemlos unter einem separaten Systemuser ausgeführt werden kann. Dies hat den Vorteil, dass besseres Performance-Tuning möglich ist, wenn Webserver und Python-Interpreter nicht eng miteinander verknüpft sind.

## 3. Python-Webframeworks

### 3.1. Vergleichskriterien

Um die Frameworks miteinander vergleichen zu können, müssen einige Kriterien festgelegt werden, anhand derer sich alle Frameworks messen lassen.

#### Modularität

Gerade bei einer großen Anwendung wie Indico gibt es diverse Bereiche, die relativ unabhängig voneinander sind. Ein Framework, das es dem Entwickler erlaubt, die Anwendung in einzelne Module aufzuteilen, ist hilfreich, um den Code besser zu strukturieren. Sofern diese Module auch verschiedene Namespaces bereitstellen, wird zudem das Risiko von Namenskollisionen verhindert.

#### URL-Routing

In modernen Webanwendungen erwartet man in der Regel saubere, semantische URLs. Dabei handelt es sich um URLs, aus deren Pfad weder Rückschlüsse auf die verwendete Technologie (z.B. *help.php* oder *conferenceDisplay.py*) gezogen werden können noch ausschließlich nichtssagende Parameter enthalten sind. Ein einfaches Beispiel für solche Parameter ist eine URL dieser Form: */view.py?type=3&id=12345*. Während *view* zwar aussagt, dass irgendein Objekt angezeigt wird und dieses Objekt die interne Identifikationsnummer *12345* besitzt, kann man der URL nicht ansehen, worum es sich dabei genau handelt. Im Gegensatz dazu steht eine URL der Form */view/event/12345-welcome*. An dieser sieht man sofort, dass es sich um ein Event handelt, welches höchstwahrscheinlich den Titel „welcome“ hat.

Um solche URLs zu realisieren, arbeiten Frameworks normalerweise mit einer Routingtabelle, die URLs auf Funktionen, Klassen, o.ä. mappt. Der Aufbau der URLs, insbesondere wenn dynamische Parameter enthalten sind, variiert von Framework zu Framework. So bieten sich reguläre Ausdrücke für maximale Flexibilität an, allerdings sind einfache Platzhalter in den meisten Fällen ausreichend und sorgen für lesbarere URLs im Code.

### **Templateengine**

Trotz der Nutzung von AJAX besteht Bedarf an dynamisch generierten HTML-Seiten. Eine Templateengine bietet dazu eine spezielle Syntax, mit deren Hilfe sowohl HTML-Dateien als auch beliebige andere textbasierte Dateiformate durch Variablen und einfache Kontrollstrukturen erweitert werden können. Während es prinzipiell möglich ist, die Templateengine zu wechseln und die meisten Templates automatisiert in die Syntax der neuen Engine zu konvertieren, hat es Vorteile, wenn das Framework mit einer beliebigen Templateengine benutzt werden kann.

### **Datenbankanbindung**

Viele Frameworks benötigen eine Datenbank, um frameworkspezifische Daten abzuspeichern oder um für den möglicherweise vorhandenen Administrationsbereich eine Benutzer- und Rechteliste zu führen. Da die in Indico genutzte ZODB nicht sehr verbreitet ist, ist insbesondere darauf zu achten, ob zur Nutzung des Frameworks eine separate Datenbank benötigt wird.

### **Sessions**

Sowohl beim Login eines Benutzers als auch bei der Nutzung diverser Funktionen von Indico müssen Daten zwischengespeichert werden. Dazu ist es praktisch, wenn das Framework bereits ein Sessionssystem bietet.

### **Caching**

Insbesondere bei häufig aufgerufenen Seiten und komplexen Datenbankzugriffen kann die Performance einer Anwendung massiv gesteigert werden, wenn entweder ganze Seiten oder Teile der dort angezeigten Daten in einem Cache abgelegt werden. Ein Framework mit einer integrierten Caching-Lösung kann dabei behilflich sein, indem es bestimmte Seiten automatisch cacht und dabei Parameter und evtl. vorhandene Berechtigungen des Benutzers beachtet.

### **Integrierbarkeit**

Bei Indico handelt es sich um eine komplexe Anwendung mit vielen Funktionen, sodass es nicht praktikabel ist, wenn der gesamte Code geändert oder in großen Teilen neu geschrieben werden muss. Daher ist bei dem neuen Framework darauf zu achten, dass es möglichst leicht in eine bestehende Anwendung integriert werden kann.

### **Erweiterbarkeit**

Unter Umständen ist es notwendig, das Framework durch zusätzliche Funktionalität zu erweitern oder bestehende Funktionen anzupassen. Dies kann über

eine Plugin-API, durch Subclassing oder aber durch Forking des Frameworks geschehen. Letzteres bedeutet, den Code des Frameworks selbst zu verändern und regelmäßig auf den aktuellen Stand der offiziellen Versionen zu bringen.

### Sonstige Features

Viele Frameworks haben neben den üblichen Features zusätzliche Funktionen, die in anderen Frameworks nicht vorhanden sind. Je nach Feature können diese für Indico nützlich sein.

### Performance

Die unterschiedlichen Frameworks sind bei der Abarbeitung von Requests möglicherweise unterschiedlich schnell. Dies ist darauf zurückzuführen, dass komplexere Frameworks mehr Dinge implizit bei jedem Request tun, während dieselbe Funktionalität in einem kompakten Framework entweder überhaupt nicht enthalten oder optional ist. Allerdings spielt dieser Performanceaspekt im Rahmen dieser Arbeit nur eine untergeordnete Rolle, da Indico zwar große Datenmengen enthält und dabei performant sein muss, jedoch der Overhead des Frameworks verglichen mit Datenbankzugriffen minimal ist und somit für den Endbenutzer in der Regel nicht spürbar ist.

### Dokumentation

Da mehrere Entwickler mit dem Framework arbeiten müssen und starke Fluktuation herrscht, da oftmals Studenten nur für kurze Zeit (6 bis 12 Monate) an Indico arbeiten, ist eine gute Dokumentation wichtig, da es nicht förderlich ist, wenn man erst den Quellcode des Frameworks lesen und verstehen muss, um es benutzen zu können. Insbesondere ist es hilfreich, wenn die Dokumentation Beispielcode enthält und *Best Practices* beschreibt, statt nur die APIs zu dokumentieren.

### Lizenz

Die meisten Frameworks sind unter einer Open-Source-Lizenz verfügbar. Da Indico unter der GNU GPL<sup>18</sup> steht, ist auf Kompatibilität mit dieser Lizenz zu achten. Abbildung 2 bietet einen kurzen Überblick über die verbreitetsten Open-Source-Lizenzen und zeigt die Kompatibilität: „To see if software can be combined, just start at their respective licenses, and find a common box you can reach following the arrows.“ [Whe07]

Da Indico unter der GPLv3 lizenziert ist, sind fast alle Open-Source-Lizenzen

---

<sup>18</sup><http://www.gnu.org/licenses/gpl-3.0.txt>

außer der *Affero GPL*<sup>19</sup> kompatibel. Die einzige Ausnahme wären Frameworks, die ausschließlich unter der GPLv2 lizenziert sind. Da die GPL jedoch bei Frameworks selten genutzt wird und insbesondere die *GPLv2-only*-Option allgemein nicht sehr verbreitet ist, ist dieses Risiko eher gering.

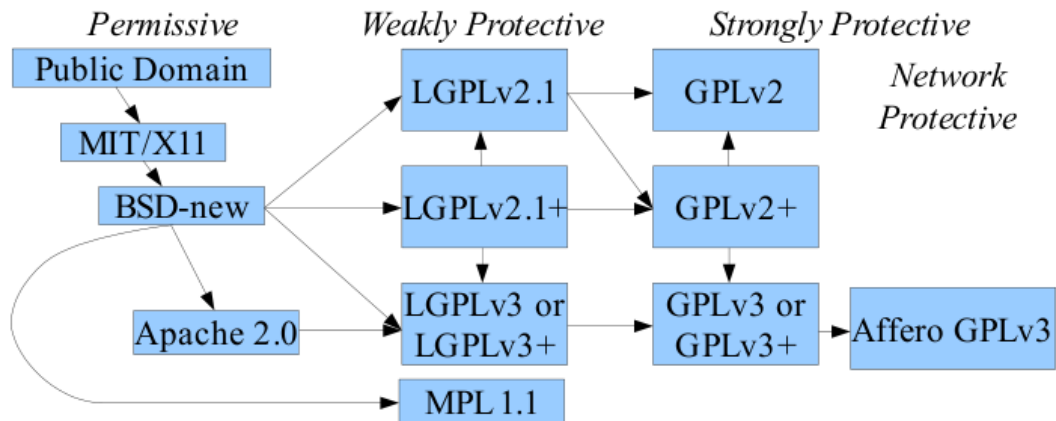


Abbildung 2: Kompatibilitätsdiagramm div. Open Source-Lizenzen [Whe07]

## 3.2. Indico

Das derzeit in Indico verwendete System kann zwar grob als Framework betrachtet werden, ist jedoch nicht wirklich ein Framework, da sehr viele Dinge speziell auf die Anwendung zugeschnitten sind, statt generisch bzw. wiederverwendbar zu sein. Dies ist bereits daran zu erkennen, dass sich der Frameworkcode nicht in einem separaten Modul oder Package befindet sondern Teil der Indico-Codebasis ist.

Den Kern des Frameworks bildet das Package `indico.web.wsgi`, welches das in Abschnitt 2.2.2 beschriebene WSGI-Interface implementiert und eine Emulationsschicht für `mod_python` bereitstellt. Neben dem Bereitstellen der entsprechenden APIs und Objekte ist die wichtigste Aufgabe dieser Schicht, Requests auf die entsprechenden Python-Dateien zu mappen. Der Aufruf von `/index.py/foo` führt zum Aufruf der Funktion `foo()` in der Datei `index.py` im `htdocs`-Ordner. Da diese Dateien jedoch, um das Verhalten von `mod_python` beizubehalten, erst dann geladen werden, wenn ein entsprechender Request empfangen wird, können sie nicht beim Initialisierung der Anwendung importiert und in einem Mapping abgelegt werden.

<sup>19</sup>GPL mit der Erweiterung, dass auch bei einem Netzwerkzugriff auf das laufende Programm eine „Verbreitung“ stattfindet und der Quellcode zugänglich gemacht werden muss.

Stattdessen wird der gesamte Code der entsprechenden Datei in einen String geladen und danach ausgeführt. Listing 10 zeigt eine vereinfachte Version des dafür zuständigen Codes.

---

```
1 def mp_legacy_publisher(req, module, handler):
2     the_module = open(module).read()
3     module_globals = {}
4     exec(the_module, module_globals)
5     return module_globals[handler](req, **req.form)
```

---

**Listing 10:** Laden der Legacy-Python-Dateien

Bekanntermaßen ist es schlechter Stil, Code aus Strings auszuführen. In Python hat es darüberhinaus noch den Nachteil, dass Debugger keinen Zugriff auf den Code haben, da sie nicht wissen, aus welcher Datei der Code stammt, und Python den Quelltext selbst nicht zusammen mit dem Bytecode im Speicher hält. In Indico ist dies allerdings kein größeres Problem, da die Dateien keine Logik enthalten sondern nur die entsprechende Handlerfunktion aufrufen. Unsauber ist es dennoch.

Neben diesen Legacy-Handlern, die jedoch für fast alle Bestandteile von Indico genutzt werden, existiert ein sehr einfaches Routingsystem, das Anfragen anhand des ersten Pfadsegments der URL an eine Handlerfunktion übergibt. Dies ist prinzipiell besser, da so die Nutzung von `exec` vermieden werden kann. Letztendlich wird jedoch genau dieselbe zuvor schon beschriebene Funktion verwendet, was leicht dazu verleitet, in der in Listing 11 gezeigten Routingtabelle direkt auf die Handlerfunktion mit der eigentlichen Logik zu verweisen.

---

```
1 {'': ((self.htdocs_dir, 'index.py'), 'index', '', None),
2  'services': ((DIR_SERVICES, 'handler.py'), 'handler', '',
3               None),
3  'export': ((DIR_MODULES, 'wsgi_handler.py'), 'handler',
4             '', None),
4  'api': ((DIR_MODULES, 'wsgi_handler.py'), 'handler', '',
5          None)
5 }
```

---

**Listing 11:** Einfaches URL-Routing



Dies führt dann dazu, dass das mit `exec` verbundene Debuggingproblem plötzlich auch relevanten Code betrifft statt nur einen einfachen Funktionsaufruf. Umgangen werden kann das zum Glück sehr einfach, indem ein separates Python-Modul nur den Aufruf der eigentlichen Handlerfunktion enthält und dieses Modul in der Routingtabelle referenziert wird.

Für die eigentliche Anwendungslogik sind die RH-Klassen zuständig. Die Basisklasse enthält dabei die Logik für die Datenbankverbindung und zum Abfangen von Fehlern, während die Subklassen davon die eigentliche Anwendungslogik implementieren. Die Abgrenzung zwischen Framework und Anwendung wird teilweise verwässert, da Teile der Basisklasse auf Anwendungscode zugreifen während diverse andere nur Frameworkcode (in der Regel das `req`-Objekt mit den `mod_python`)-Daten nutzen.

### Modularität

Indico enthält ein Modulsystem, welches sowohl für Code, der ein fester Bestandteil von Indico ist, als auch für externen Code genutzt werden kann. In beiden Fällen bietet das Modulsystem einen Namespace in der Datenbank, der ausschließlich vom jeweiligen Modul genutzt wird, und einen Menüpunkt in der Administrationsoberfläche, über den das Modul aktiviert bzw. deaktiviert und konfiguriert werden kann. Dies hat den großen Vorteil, dass die Konfiguration bei den meisten Modulen sehr einfach ist - sowohl für die Administratoren als auch für den Entwickler.

Einige Module (z.B. das Interface zum CERN-Paymentsystem und zur CERN-Suchmaschine) sind nicht Teil des Open-Source-Projekts. Daher muss es für solche Module eine Möglichkeit geben, auch ohne direkt im Indico-Code referenziert zu werden, auf Datenstrukturen zugreifen und bei bestimmten Ereignissen Code ausführen zu können. Dies wurde in Indico mithilfe von *entry points* realisiert. Dabei handelt es sich um ein Feature des Paketsystems von Python, über das ein Paket Objekte für einen bestimmten *entry point* zentral registrieren kann. Indico kann dann unter Verwendung der Funktion `pkg_resources.iter_entry_points('indico.ext')` über diese Objekte iterieren und die entsprechenden Plugins laden.

Plugins können über reguläre Ausdrücke Code für bestimmte URLs ausführen, sodass sie auch eigene Seiten hinzufügen können; in der Regel folgt die URL-Struktur dabei den `mod_python`-URLs, d.h. *something.py* bzw. *something.py/someAction*. Daneben können Plugins sowohl Funktionen für die JSON-RPC-API als auch Endpoints für die REST-basierte Export-API registrieren.

## URL-Routing

Wie oben bereits beschrieben, emuliert Indico *mod\_python* und verwendet daher das bereits aus CGI-Anwendungen Mapping zwischen URLs und Dateisystem. Der WSGI-Kern von Indico kann Anfragen anhand des ersten Pfadsegments routen, allerdings ist dies für die wenigsten Use-Cases ausreichend. Für *clean URLs* sind in der Regel alle Pfadsegmente relevant und es müssen dynamische Segmente unterstützt werden, um IDs und sonstige Parameter in den Pfad einbauen zu können.

Indico-Plugins haben die Möglichkeit, über einen regulären Ausdruck beliebige URLs zu mappen und über Regex-Gruppen auch auf einzelne Pfadelemente zuzugreifen, um z.B. eine ID zu extrahieren. Dieses Feature ist jedoch ausschließlich in Plugins verfügbar und nicht für Code, der Teil des Indico-Kerns ist. Zusätzlich müssen die regulären Ausdrücke jeweils in der *RH*-Klasse definiert werden, wodurch die URL-Mappings über den gesamten Plugincode verstreut sind statt übersichtlich in einer einzelnen Datei aufgelistet zu sein.

Die Export-API nutzt ebenfalls reguläre Ausdrücke, um URLs unterhalb von */export/* und */api/* auf die jeweiligen Klassen zu mappen. Allerdings ist dieses Routing speziell auf die Struktur der API zugeschnitten und weder Teil des eigentlichen Indico-Frameworks noch ohne Weiteres für andere Bereiche von Indico nutzbar.

Unabhängig davon, wie eine bestimmte URL auf den dazugehörigen Code gemappt wird, bietet Indico mit den *URLHandler*-Klassen eine Möglichkeit, URLs zentral zu definieren und zu generieren. Dabei können beliebige GET-Parameter hinzugefügt werden, wobei die jeweilige *URLHandler*-Klasse diese modifizieren kann, bevor sie an die URL angehängt werden. Die Änderung einer URL im *URLHandler* wirkt sich jedoch logischerweise nur auf die Generierung von URLs und nicht auf das Routing auf, was dem DRY-Konzept<sup>20</sup> widerspricht.

## Templateengine

Indico nutzt die Mako-Templateengine<sup>21</sup>. Dabei handelt es sich um eine von Indico unabhängige Open-Source-Templateengine unter der MIT-Lizenz. Sie unterstützt alle von einer modernen Templateengine erwarteten Features wie Vererbung, Includes, Makros, Blöcke, Variablen, Filter und Kontrollstruktu-

---

<sup>20</sup>Don't Repeat Yourself

<sup>21</sup><http://www.makotemplates.org>

ren. Daneben erlaubt sie auch, beliebigen Python-Code in einem Template zu verwenden.

Mako wird in Indico unverändert verwendet, allerdings wird der Template-kontext um diverse Hilfsfunktionen und -variablen erweitert, sodass häufig genutzte Funktionen in jedem Template verfügbar sind.

```
1 <%inherit file="Footer.CERN.tpl" />
2 <%block name="footer">
3 % if showSocial:
4     <%include file="SocialIcons.tpl"
5         args="dark=dark,url=shortURL"/>
6 % endif
7     ${parent.footer()}
8 </%block>
```

---

**Listing 12:** Mako-Template

### Datenbankanbindung

Indico enthält ein einfaches Interface zum Zugriff auf die ZODB-Datenbank. Dieses wird unter anderem in den RH-Klassen und den verschiedenen APIs genutzt, um eine Datenbankverbindung aufzubauen und im Fehlerfall die Ausführung des Requests zu wiederholen. Während man das Datenbankinterface selbst definitiv zum Framework zählen kann, ist der Code für die Wiederholungen stark mit Anwendungscode vermischt.

Das Modulsystem von Indico nutzt die Datenbank sowohl zum Speichern der Konfigurationseinstellungen als auch für die Metadaten der verschiedenen Module, sodass sie nicht bei jedem Aufruf neu über die *entry points* geladen werden müssen. Dabei wird massiv darauf gesetzt, dass es sich bei ZODB um eine Python-Objektdatenbank handelt. Damit können auch komplexe Datentypen wie Python-Module direkt in der Datenbank gespeichert werden statt nur den Pfad zur dazugehörigen Python-Datei abzulegen.

### Sessions

Indico speichert Sessions in der ZODB-Datenbank ab. Dies erlaubt es, neben beliebigen anderen Objekten, auch bereits in der Datenbank gespeicherte Objekte in der Session abzulegen, ohne zusätzlichen Speicher zu benötigen.

Zum Speichern von Daten in der Session existieren zwei Möglichkeiten. Häufig genutzte Elemente wie der aktuelle User oder seine Spracheinstellung sind direkt im Sessionobjekt abgelegt und dementsprechend über ihre Gettermethode oder direkt über die darunterliegende Eigenschaft des Objekts abrufbar. Darüberhinaus können beliebige Daten ähnlich wie in einem `dict` abgelegt werden. Dies geschieht über die Methoden `setVar()`, `getVar()` und `removeVar()`.

Eine Session wird nur dann in der Datenbank abgespeichert, wenn sie auch Daten enthält. Dies hat den Vorteil, dass insbesondere für nicht eingeloggte Benutzer kein Sessionobjekt in der Datenbank abgespeichert werden muss. Nichtsdestotrotz sammeln sich mit der Zeit sehr viele alte Session an; mangels einer Möglichkeit, einzelne Objekte in der ZODB automatisch nach einer gewissen Zeit zu verwerfen, müssen alte Sessions regelmäßig manuell bzw. über ein zeitgesteuertes Script aus der Datenbank gelöscht zu werden.

Während eines Requests, wird die Session als Eigenschaft im *mod\_python*-Request-Objekt gespeichert. Das für Sessions zwingend notwendige Cookie wird ebenfalls über dieses Objekt gesetzt. Dies hat zur Folge, dass das Sessionssystem stark von der *mod\_python*-Emulation abhängig ist.

## Caching

Indico bietet ein generisches Cache-Interface, das verschiedene Backends nutzen kann. Standardmäßig werden Memcached, Redis und Dateien unterstützt, wobei es für den Entwickler auch einfach ist, weitere Backends hinzuzufügen. Das Interface beschränkt sich auf die am häufigsten genutzten Methoden `get`, `set` und `delete`, wobei jeweils auch eine `_multi`-Variante existiert, die je nach Backend performanter ist sofern mehrere Einträge auf einmal verändert bzw. ausgelesen werden sollen. Ebenfalls unabhängig vom Backend kann für jeden Cache-Eintrag eine Expire-Zeit angegeben werden. Dies hat bei der Nutzung des Datei-Backends jedoch zur Folge, dass abgelaufene Einträge nur gelöscht werden, wenn versucht wird, auf sie zuzugreifen.

Zur besseren Strukturierung des Codes kann jeder Instanz des `GenericCache`-Objekts ein Namespace zugewiesen werden, der automatisch mit dem jeweiligen Cache-Key kombiniert wird. Dies hat ebenfalls den Vorteil, dass nicht eine einzelne Instanz der Klasse global verfügbar sein muss sondern z.B. in einer Klasse, die den Cache nutzen will, eine neue Instanz erstellt wird. Bei nicht-lokalen Caches wie Redis und Memcached wird dabei instanzübergrei-

fend eine bestehende Verbindung verwendet, sodass auch bei vielen Instanzen nicht übermäßig viele Verbindungen aufgebaut werden müssen.

### **Integrierbarkeit**

Da das derzeitige Framework ein Teil von Indico ist, ist dieser Punkt nicht relevant. Der Vollständigkeit halber sei erwähnt, dass es nicht praktikabel wäre, das Indico-Framework von der Indico-Anwendung zu trennen um es in einer anderen Anwendung nutzen zu können.

### **Erweiterbarkeit**

Als Bestandteil von Indico ist die einfachste Art, das Framework zu erweitern, es direkt zu modifizieren. Da keine Standalone-Version existiert, ist kein Fork notwendig, weshalb auch die damit üblicherweise verbundenen Probleme ausbleiben.

### **Sonstige Features**

Indico bietet mit dem `ContextManager` eine Möglichkeit, threadsicher Daten ähnlich wie in einer globalen Variable abzuspeichern. Diese werden jeweils zu Beginn eines neuen Requests gelöscht, sodass dort temporäre Daten abgelegt werden können, ohne sie als Funktionsparameter übergeben zu müssen. Insbesondere für Dinge wie das aktuelle Request-Objekt hat dies den Vorteil, dass es falls benötigt jederzeit verfügbar ist, ohne immer nicht explizit übergeben zu werden.

### **Dokumentation**

Es existiert keine echte externe Dokumentation. Der Code ist teilweise kommentiert, wobei der Kommentarstil nicht einheitlich ist und man kann deutlich erkennen, dass mehrere Entwickler daran gearbeitet haben. Dementsprechend sind einige Teile sehr ausführlich kommentiert, während andere fast überhaupt nicht kommentiert sind.

Die Dokumentation im Indico-Wiki<sup>22</sup> geht hauptsächlich auf anwendungsspezifische Dinge ein, weshalb sie insbesondere auch für einen neuen Entwickler, der etwas am Framework selbst ändern muss, nicht sehr hilfreich ist.

### **Lizenz**

Da das Framework Teil von Indico ist und auch nicht *standalone* verfügbar ist, steht es genau wie Indico selbst unter der GPL.

---

<sup>22</sup><http://indico-software.org/wiki/Dev>

### 3.3. Django

*Django*<sup>23</sup> ist das wohl bekannteste und funktionsreichste Full-Stack-Webframework für Python. *Full-Stack* bedeutet dabei, dass alle für eine typische Webanwendung wichtigen Komponenten wie Datenbankzugriff, Routing, Templates und Sessions vom Framework bereitgestellt werden.

Es ist primär auf *Rapid Application Development* ausgelegt und wird daher von den Entwicklern auch mit dem Slogan „for perfectionists with deadlines“ beworben. Wie für RAD-Frameworks typisch bietet Django über das Management-Tool *django-admin.py* diverse Scaffolding-Funktionen, d.h. die Möglichkeit, die Grundstruktur für ein neues Projekt bzw. neue Module innerhalb eines Projekts automatisch zu generieren.

Die Entwicklung von Django begann 2003, wobei die erste Open-Source-Version 2005 veröffentlicht wurde. Seitdem wurde das Framework konstant weiterentwickelt und ist derzeit in der Version 1.6 verfügbar, die unter anderem Kompatibilität mit Python 3 bietet.

#### Modularität

Django bietet Modularität in verschiedenen Schichten. Bei der Webanwendung, die der Benutzer sieht handelt es sich - um bei der in der Django-Dokumentation genutzten Terminologie zu bleiben - um das *Project*. Dabei handelt es sich in der Regel um ein in sich abgeschlossenes Projekt, das die Konfiguration sowohl für Django selbst, die genutzte Datenbank und die Anwendung enthält. Ein *Project* enthält eine oder mehrere *Apps*, wobei es sich bei jeder App um ein Python-Paket handelt, das Datenbankmodelle, Viewfunktionen, Templates und Anwendungslogik enthält. Die Anwendung auf viele kleine Apps aufzuteilen hat den Vorteil, dass in sich abgeschlossene Funktionalität wie Tagging oder auch ein Loginsystem sauber von dem restlichen Projekt abgegrenzt ist und damit in der Regel wiederverwendbar ist. Dies wird von Django insofern unterstützt, dass auch problemlos Apps aus anderen Projekten eingebunden werden können und jede App in der Regel unabhängig von ihrem *Project* ist sofern die nötigen Konfigurationsdaten vorhanden sind. Django selbst enthält diverse Apps für optionale Zusatzfunktionen wie eine Benutzerverwaltung und Kommentarfunktion.

---

<sup>23</sup><https://www.djangoproject.com>

Unabhängig von den installierten Apps bietet Django den Middleware-Stack, über den auf einer relativ niedrigen Ebene in die Abarbeitung von Requests eingegriffen werden kann. `process_request` ermöglicht es der Middleware, vor dem Routing eines Requests einzugreifen und entweder die reguläre Verarbeitung fortsetzen oder sie mit einer HTTP-Antwort vorzeitig beenden. Dies bietet sich z.B. an, um Seiten anhand der aufgerufenen URL zu cachieren. `process_view` verhält sich ähnlich, allerdings ist bei der Ausführung dieses Middleware-Hooks bereits bekannt, welche Viewfunktion den Request abarbeiten wird.

Middleware hat außerdem die Möglichkeit, nach der Verarbeitung eines Requests einzugreifen. Dazu stehen die Methoden `process_response` und `process_exception` zur Verfügung. Diese könnten z.B. einen Cache aktualisieren oder einen webbasierten Debugger starten.

### URL-Routing

Django legt großen Wert auf saubere URLs, die weder Dateieindungen noch kryptische Elemente wie `/0,2097,1-1-1928,00` enthalten. Aus diesem Grund bietet Django ein flexibles und mächtiges Routingsystem. Da der Kern des Routingsystems der Vergleich von definierten Mustern mit der aufgerufenen URL ist, bieten sich reguläre Ausdrücke perfekt an und werden dementsprechend auch verwendet.

Der Entwickler definiert in seiner App dazu eine Liste mit *Patterns*, wobei jedes Element aus einem regulären Ausdruck und der Viewfunktion, wobei es sich bei letzterem entweder direkt um die Funktion handeln kann oder aber um einen String der Form `'myapp.views.somefunc'`. Für dynamische Elemente in der URL wird das Gruppierungs-Feature der Regex-Engine verwendet. Dies bedeutet, dass der Ausdruck `r'news/(\d+)` eine URL nach dem Schema `news/123` matchen würde und die Zahl als positionalen Parameter an die Viewfunktion übergibt. Da dies gerade in komplexeren URLs mit mehreren Parametern schnell unübersichtlich würde, können auch Namen für die Parameter in der Form `r'news/(?P<id>\d+)` vergeben werden. Diese werden dann als Keyword Arguments an die Viewfunktion übergeben.

Es ist zu erwähnen, dass das Routing ausschließlich die regulären Ausdrücke benutzt und somit nur den Pfad in der URL berücksichtigt. Das HTTP-Verb, also meist *GET* oder *POST* und der Domainname sind also nicht Teil des Routings.

Insbesondere mit Apps, die zusätzliche Funktionalität wie eine Kommentarfunktion bereitstellen, würde es dem Prinzip der sauberen URLs widersprechen, alle URLs auf der Rootebene zu registrieren. Dazu bietet Django die `include()`-Funktion, mit der anstelle auf eine Viewfunktion auch auf eine andere URL-Routingtabelle verwiesen werden kann.

Um anhand der Viewfunktion oder eines optionalen Identifiers eine URL zu generieren, bietet Django sowohl die Python-Funktion `reverse()` als auch eine spezielle Syntax für Templates. Ganz im Sinne von DRY führt dies dazu, dass URLs niemals manuell erzeugt werden müssen und Änderungen keine toten internen Links zur Folge haben, da Dispatching und URL-Generierung beide auf dieselben Routingtabellen zugreifen.

### Templateengine

Django verwendet eine eigene Templateengine, die speziell für das Framework entwickelt wurde und dementsprechend gut integriert ist. Sie unterstützt alle von einer modernen Templateengine erwarteten Features und ermöglicht direkten Zugriff auf das Routingsystem bzw. die dazu gehörende URL-Generierung von Django. Aus Sicherheitsgründen werden alle dynamischen Daten in HTML-Templates escaped, sofern es nicht explizit deaktiviert wird.

Da es sich bei Templates offensichtlich um die Präsentationsschicht handelt und dort höchstens ausgabespezifische Logik vorhanden sein soll, ist es in Django-Templates nicht möglich, Python-Code direkt einzubinden. Allerdings können Python-Funktionen in Templates zugänglich gemacht werden und beliebige Methoden von übergebenen Python-Objekten ausgeführt werden.

```
1 {% extends 'base.html' %}
2 {% block title %}Beispiel{% endblock %}
3 {% block content %}
4     <a href="{% url 'example' example.id %}">
5         {{ example.title }}
6     </a>
7 {% endblock %}
```

**Listing 13:** Django-Template



## Datenbankanbindung

Django enthält ein ORM-System, welches die meisten verbreiteten relationalen Datenbanksysteme wie PostgreSQL, MySQL, Oracle und SQLite offiziell unterstützt und über inoffizielle Zusatzmodule auch weitere Datenbanken wie den Microsoft SQL Server nutzen kann.

Während es bei einer neuen Anwendung üblich ist, die Model-Klassen für die Datenbanktabellen manuell zu erstellen und im Anschluss auf diesen basierend die Tabellen zu erstellen, bietet Django auch die Möglichkeit, eine existierende Datenbank zu analysieren und die dazu passenden Klassen zu generieren.

Teile von Django selbst benötigen eine mit dem Django-ORM kompatible SQL-Datenbank. Allerdings gibt es die Django-Erweiterung *Django-ZODB* um die ZODB für Anwendungsdaten nutzen zu können.

## Sessions

Das Session-System von Django ist zwar Teil des `django`-Pakets, allerdings ist es nicht Teil des Django-Kerns sondern als App und Middleware realisiert, was ein gutes Beispiel für die Modularität des Frameworks ist. Wenn es aktiviert ist stellt es in `request.session` ein dict-artiges Objekt bereit, über das auf die Daten der Session zugegriffen werden kann. Bei `request` handelt es sich um den an jede Viewfunktion übergebenen Parameter, über den auf alle zum aktuellen Request gehörenden Daten zugegriffen werden kann.

Die meisten Eigenschaften der Session-App können konfiguriert werden was sie sehr flexibel macht. Insbesondere können verschiedene Storage-Backends verwendet werden - u.a. einen Cache wie *memcached*, signierte Cookies oder eine Tabelle in der Datenbank - und man hat die Wahl zwischen *Pickle*, *JSON* oder einer eigenen Implementierung bei der Serialisierung der Sessiondaten. Letzteres ist insbesondere bei clientseitigen Cookie-Sessions wichtig, da es sich bei *Pickle* zwar um ein sehr mächtiges Format handelt, das fast jeden Python-Datentyp serialisieren kann, diese Flexibilität jedoch zur Folge hat, dass beim Deserialisieren auch beliebiger Code ausgeführt werden kann. Durch die kryptografische Signatur kann ein Benutzer zwar prinzipiell keine böartigen Daten als gültiges Sessioncookie übermitteln, allerdings führt ein Leak des geheimen Schlüssels sofort auch zu einer *Remote Code Execution*-Lücke, da mit diesem Schlüssel gültige Cookiesignaturen erstellt werden können.

Seitens der Entwickler wird das Cache-Backend empfohlen; in diesem Fall bietet sich der *Pickle*-Serializer an, da bei dieser Kombination sowohl die hohe

Performance des Caches als auch die Flexibilität von Pickle zur Verfügung stehen und der Client niemals mit den Sessiondaten in Berührung kommt und somit keine Möglichkeit hat, sie zu manipulieren.

Das Sessionmodul nutzt unabhängig vom Backend ein Cookie; entweder zum Speichern des einzigartigen Session-Identifiers oder für die Daten selbst. Dies hat zur Folge, dass das Session-System bei deaktivierten Cookies nicht funktioniert. Dies ist heutzutage allerdings kein relevantes Problem mehr, da fast jede Website spätestens beim Login Cookies voraussetzt.

## Caching

Wie zuvor schon erwähnt, eignet sich Djangos Middleware-System unter anderem ideal dafür, ganze Seiten zu cachen. Dementsprechend ist es nur logisch, dass Django eine entsprechende Middleware mitliefert. Diese realisiert Caching wie zu erwarten auf Seitenebene, d.h. jede via GET aufgerufene Seite wird gecacht, sofern es nicht durch entsprechende Header unterbunden wird. Insbesondere auf Seiten, deren Content nicht extrem oft aktualisiert wird, bietet sich diese Cachemethode an, da sie mit minimalem Aufwand realisierbar ist.

Eine flexiblere Cachemethode ist der Decorator-basierte View-Cache. Dieser ermöglicht es, einzelne Views zu cachen, sodass z.B. die Startseite gecacht werden kann aber andere, häufiger aktualisierte Seiten wie ein Gästebuch, nicht gecacht werden. Sofern die Granularität immer noch nicht ausreicht, erlaubt Django es auch, einzelne Templatefragmente zu cachen. Dies bietet sich an, wenn Teile eines Templates zu dynamisch zum cachen sind, andere jedoch nur relativ selten aktualisiert werden.

Neben diesen Methoden bietet Django auch eine Lowlevel-API, die direkten Cachezugriff ermöglicht. Diese unterstützt mehrere Caches, die auch verschiedene Backends nutzen können, und bietet neben den üblichen Methoden `get`, `set`, `delete` und den Varianten für mehrere Keys auch `incr` und `decr` zum Inkrementieren bzw. Dekrementieren von numerischen Werten. Dabei handelt es sich, sofern es vom Backend unterstützt wird, um atomare Operationen.

Standardmäßig werden Memcached, Datenbanktabellen, Dateien und ein insbesondere während der Entwicklung hilfreicher prozess-lokaler In-Memory-Cache unterstützt. Es ist auch möglich, ein benutzerdefiniertes Cache-Backend zu verwenden.

### Integrierbarkeit

Da es sich bei Django um ein Full-Stack-Framework handelt ist es darauf ausgelegt, von Anfang an verwendet zu werden. Es ist zwar prinzipiell möglich, Teile von Django, wie das Routingsystem, zu nutzen, ohne die fortgeschritteneren Features, wie das ORM, zu verwenden, allerdings muss man dabei in der Regel auf viele Vorteile des Frameworks verzichten. So müssten z.B. alle Templates der bestehenden Anwendung angepasst werden, um die URL-Generierung des Routingsystems zu verwenden.

Explizit unterstützt wird jedoch die Nutzung einer bereits existierenden Datenbank: Wie bereits erwähnt, kann Django die Model-Klassen anhand des Schemas der Datenbank generieren. Somit spart man sich den Aufwand, die Model-Klassen zu erstellen, und kann sich darauf konzentrieren, die Anwendungslogik neu zu implementieren.

### Erweiterbarkeit

Der einfachste Weg, Django zu erweitern, ist durch Middleware. Sofern es nicht ausreicht, vor und nach der Abarbeitung eines Requests Code auszuführen, bietet Django mit *Signals* ein Callback-System, welches insbesondere eine Reaktion auf verschiedene Datenbankereignisse ermöglicht.

Um das Verhalten des Django-Kerns selbst zu verändern, ist es jedoch in der Regel notwendig, den Frameworkcode zu modifizieren. Dies bedeutet, dass entweder ein Fork notwendig ist oder aber dass die Änderungen allgemein genug sind, um eine Chance zu haben, von den Django-Entwicklern in die offizielle Django-Version übernommen zu werden.

### Sonstige Features

RAD-typisch kann Django für einfache CRUD-Aufgaben in der Administration einer Website sowohl die Logik als auch das Benutzerinterface dynamisch generieren. Dabei handelt es sich nicht um das in vielen Frameworks übliche Scaffolding, bei dem der entsprechende Code einmalig generiert und danach modifiziert wird. Stattdessen werden die Formulare anhand der Datenbankmodelle dynamisch generiert, wobei dieser Vorgang vom Entwickler beeinflusst werden kann, um auch komplexere Elemente wie One-To-Many-Beziehungen in der gewünschten Art und Weise in einem Formular repräsentieren zu können.

Auch das wohl wichtigste Feature in den meisten Webapplikationen, die Authentifikation von Benutzern, wird von Django standardmäßig unterstützt, sofern man die entsprechende App aktiviert. Neben der Benutzerverwaltung

selbst und der Authentifikation mittels eines Passworts oder über einen Drittanbieter enthält das Benutzersystem ein Gruppen- und Rechtesystem und diverse Decorators um den Zugriff auf einzelne Views zu beschränken.

### Dokumentation

Django besitzt eine sehr ausführliche Online-Dokumentation sowohl für die aktuelle als auch für ältere Versionen, die jeweils auch in verschiedenen Formaten heruntergeladen werden kann. Neben einer allgemeinen Beschreibung der verschiedenen Bestandteile des Frameworks enthält sie auch ein Step-By-Step-Tutorial und eine Beschreibung aller in Django genutzten Klassen und Funktionen.

Der Code von Django ist größtenteils sauber dokumentiert bzw. kommentiert, wobei leider im Django-Kern einige interne Funktionen für einen Außenstehenden, der nicht mit dem Code vertraut ist, nicht selbsterklärend sind. Allerdings ist es in der Regel nicht notwendig oder angebracht, Frameworkcode zu verändern und dank der guten Dokumentation der APIs muss man auch nur selten etwas direkt im Code nachschauen.

### Lizenz

Django steht unter der BSD-Lizenz. Dabei handelt es sich um eine *permissive* Open-Source-Lizenz, die nur verlangt, dass der Copyrighthinweise im Code erhalten bleibt und der Name des ursprünglichen Entwicklers nicht missbräuchlich verwendet wird. Damit ist die Lizenz ideal für ein Framework geeignet, da sie der eigentlichen Anwendung keine bestimmte Lizenz aufzwingt und auch die kommerzielle Nutzung ohne Einschränkungen erlaubt.

## 3.4. Flask

*Flask*<sup>24</sup> ist ein relativ leichtgewichtiges Microframework, dessen Entwicklung 2010 begann und derzeit in Version 0.10 verfügbar ist. Neben Python 2.6 und 2.7 unterstützt die aktuellste Version des Frameworks auch Python 3.3.

Der Begriff „Microframework“ bedeutet bei Flask, dass das Framework dem Entwickler die größtmögliche Flexibilität lässt, welche Technologien er für die einzelnen Bestandteile seiner Anwendung nutzt.

---

<sup>24</sup><http://flask.pocoo.org>

Kern von Flask ist das WSGI-Toolkit *Werkzeug*, das die Lowlevel-Funktionen für das WSGI-Interface und alle HTTP-spezifischen Utilityfunktionen bereitstellt. Auch das URL-Routingsystem ist Teil von Werkzeug, wobei Flask die High-Level-APIs dazu bereitstellt und dadurch den in einem Framework erwarteten Komfort bietet.

### Modularität

Flask ermöglicht modulare Anwendungen mithilfe von *Blueprints*. Diese verhalten sich ähnlich wie eine vollwertige Flask-Anwendung, allerdings fehlt die gesamte für eine Flask-Anwendung notwendige Logik. Stattdessen fügen Blueprints ihre Bestandteile der Anwendung hinzu, sobald sie bei der Anwendung registriert werden. Im Gegensatz zu mehreren kleineren Anwendungen haben Blueprints den Vorteil, dass sie weder für die anwendungsweite Fehlerbehandlung noch für Funktionalität wie die Datenbankverbindung zuständig sind, sondern all diese Dinge von der Anwendung selbst übernehmen. Ein Blueprint kann Templates, statische Daten, URL-Routingdaten und Viewfunktionen enthalten. Darüber hinaus ist es möglich, Exceptions auf Blueprintebene abzufangen statt sie grundsätzlich an das Errorhandling der Anwendung selbst weiterzureichen.

Allerdings hat das Blueprint-System auch einen Nachteil: Dadurch, dass es sich gerade nicht um eine vollwertige Anwendung handelt, eignen sie sich nur bedingt dazu, komplett wiederverwendbare Module zu entwickeln, die auch von Dritten *as-is* verwendet werden können. Diesem Problem kann jedoch bei Bedarf entgegengewirkt werden, indem man entsprechende Voraussetzungen an die Anwendung stellt, die den Blueprint nutzen soll, oder problematische Teile entsprechend abstrahiert. Eine Klasse, die auf Benutzerdaten zugreift, könnte z.B. abstrakte Methoden enthalten, die in der jeweiligen Anwendung dann überschrieben werden müssen.

### URL-Routing

Wie jedes moderne Framework setzt auch Flask auf saubere URLs. Dazu nutzt es das von Werkzeug bereitgestellte URL-Routing-System und erweitert es um eine einfachere API, die insbesondere bei der Registrierung von Routingregeln mittels Decorators und beim Generieren von URLs ihre Vorteile zeigt.

Die einfachste Möglichkeit, eine Routingregel hinzuzufügen, ist mit dem Decorator `app.route`. Dieser akzeptiert als Parameter die Routingregel und optional die erlaubten HTTP-Methoden. Die damit dekorierte Funktion wird daraufhin automatisch mit der Regel verknüpft und der Funktionsname als

Endpoint benutzt. Natürlich sind in einer größeren Anwendung nur wenige URLs komplett statisch, sondern enthält auch diverse Variablen. Da es sich bei diesen in den meisten Fällen um einfache Strings oder Zahlen handelt, werden diese durch einfache Platzhalter in der Form '`<type:name>`' bzw. bei Strings '`<name>`' angegeben statt über oftmals eher kryptische reguläre Ausdrücke. Standardmäßig unterstützt Flask die Typen `string`, `int`, `float` und `path`, wobei es möglich ist, eigene Typen zu definieren indem man eine Subklasse von `BaseConverter` definiert und darin die Methoden `to_url` und `to_python` implementiert. Intern nutzt das Routingsystem reguläre Ausdrücke mit denen man allerdings nur in Kontakt kommt, wenn man einen eigenen Typkonverter definiert. Die Übergabe der Parameter an die Viewfunktion erfolgt über Keyword Arguments. Auch optionale Parameter sind möglich; in diesem Fall muss die Funktion mindestens eine Routingregel besitzen, die den entsprechenden Parameter nicht enthält.

Sofern Decorators keine Option sind - z.B. weil der Code über viele Dateien verstreut ist und man die Routingregeln an einem zentralen Ort haben will - bietet Flask die Methode `app.add_url_rule`. Diese entspricht prinzipiell dem Decorator, wobei die Viewfunktion ebenfalls als Parameter übergeben wird.

Neben dem Pfad und dem HTTP-Verb kann das Routingsystem auch die aktuelle Subdomain berücksichtigen, sowohl als festen Wert als auch als dynamische Variable. Für komplexere Fälle kann auch direkt auf die Map des Werkzeug-Routingsystems zugegriffen werden statt die Flask-API zu nutzen. Dies ist jedoch gerade dank Blueprints nur in den wenigstens Fällen notwendig.

Um über die Routingtabelle eine URL zu generieren, benötigt man den Namen des Endpoints der entsprechenden Routingregel. Übergibt man diesen an die Funktion `url_for`, kann diese eine URL daraus generieren. Standardmäßig ist diese relativ zur aktuellen Domain, also in der Form `/foo/bar`, allerdings akzeptiert die Funktion neben den Keyword Arguments für die Variablen der Routingregel auch die speziellen Argumente `_external` um eine vollständige URL zu generieren, `_schema` um das Protokoll in einer solchen absoluten URL festzulegen und `_anchor` um das URL-Fragment (`#something`) anzugeben. Sofern mehrere Regeln für denselben Endpoint existieren, verwendet `url_for` automatisch die passendste Regel.

Blueprints besitzen dieselben Methoden; der einzige relevante Unterschied ist,

dass ein Blueprint einen Präfix besitzen kann, der jeder Routingregel des Blueprints hinzugefügt wird. Bei der Nutzung von `url_for()` wird ein Blueprint in der Form `'blueprint.endpoint'` angegeben, wobei auch relative Verweise der Form `'.endpoint'` möglich sind; in diesem Fall wird der zum Zeitpunkt des Aufrufs aktive Blueprint benutzt.

### Templateengine

Flask nutzt standardmäßig die *Jinja2*-Templateengine, die auch unabhängig von Flask verfügbar ist. Die Templatesyntax ist größtenteils mit der von Django identisch und auch die Features sind sehr ähnlich. Flask stellt diverse Funktionen - insbesondere `url_for()` zum Generieren von URLs - und u.a. die Request- und Sessionobjekte in allen Templates zur Verfügung und escaped dynamische Daten in HTML-Templates automatisch, sofern es nicht explizit deaktiviert wird.

Um Jinja2 anzupassen - sei es mit benutzerdefinierten Templatefiltern oder zusätzlichen globalen Funktionen - stellt Flask im Applikationsobjekt entsprechende Funktionen zur Verfügung, die jeweils auch als Decorator verwendet werden können.

Jinja2 steht in einer Flask-Anwendung immer zur Verfügung; es handelt sich dabei um eine feste Abhängigkeit des Frameworks, die auch nicht deaktiviert werden kann. Der Sinn dahinter ist, dass Flask-Erweiterungen die Engine immer nutzen können und nicht Templates für verschiedene Engines mitliefern müssen.

Da der Zugriff auf Templates in Flask jedoch über die im `flask`-Package definierte Funktion `render_template()` geschieht, die jeweils explizit aufgerufen werden muss, steht es jedem Entwickler frei, in seiner Anwendung eine andere Templateengine zu nutzen. Allerdings muss er in diesem Fall selbst darauf achten, Dinge wie Autoescaping zu konfigurieren und die Flask-Objekte falls benötigt in Templates verfügbar zu machen. Für die *Mako*-Templateengine gibt es jedoch bereits eine Flask-Extension, die Mako sauber in Flask integriert. Insbesondere wird exakt derselbe Templatekontext verwendet, der auch an Jinja2 übergeben wird. Dies hat den Vorteil, dass der entsprechende Decorator in Flask weiterhin benutzt werden kann. Ebenfalls via Extension unterstützt wird die XML-basierte Templateengine *Genshi*.

## Datenbankanbindung

Flask selbst nutzt keine Datenbank und enthält, Microframework-typisch, keinen Datenbankcode. Es existiert mit *Flask-SQLAlchemy* jedoch eine offizielle Flask-Erweiterung, die das SQLAlchemy-ORM-System in Flask integriert und dabei denselben Komfort bietet, der auch bei allen anderen Features von Flask üblich ist. Neben den ORM-Features kann alternativ auch nur die Datenbank-abstraktionsschicht von SQLAlchemy genutzt werden.

Mit *Flask-MongoKit*, *Flask-PyMongo* und *Flask-MongoAlchemy* existieren auch verschiedene Erweiterungen, um die NoSQL-Datenbank *MongoDB* in Flask zu integrieren. Der Grund für die drei verschiedenen Extensions ist, dass je nach Anwendung ein höherer oder niedrigerer Abstraktionsgrad bei der Datenbank-API gewünscht ist.

Auch für die Objektdatenbank *ZODB* existiert eine Flask-Erweiterung, die genau wie *Flask-SQLAlchemy* den *approved extension*-Status hat und somit den Design Guidelines von Flask folgt, entsprechend lizenziert ist und eine entsprechend hochwertige Dokumentation besitzt.

## Sessions

Flask stellt über das `session`-Objekt ein spezielles `dict` bereit, welches sich für den Entwickler wie ein normales `dict` verhält, jedoch Veränderungen automatisch registriert. Daher kann es in der Regel ohne Weiteres modifiziert werden und die Daten werden automatisch in der Session abgespeichert. Neben den Standardmethoden enthält das Objekt diverse Eigenschaften, um den Status der Session auszulesen bzw. zu verändern. `new` und `modified` geben an, ob die Session noch nie gespeichert wurde bzw. ob sie seit dem letzten Speichern verändert wurde. Die `permanent`-Eigenschaft setzt die Ablaufzeit des Session-Cookies; standardmäßig ist es nur bis zum Schließen des Browsers gültig.

Da Flask weder eine Datenbank noch einen Cache voraussetzt, werden Sessions standardmäßig clientseitig als signiertes Cookie gespeichert. Dies ist eine sehr einfache aber oftmals ausreichende Lösung, sofern man weder geheime noch größere Daten abspeichern will, da Cookies oftmals auf 4096 Bytes beschränkt sind und die Signatur zwar vor Veränderungen schützt, nicht jedoch vor Auslesen.

Aus Sicherheitsgründen werden Sessiondaten standardmäßig als *JSON* serialisiert statt das mächtigere *Pickle* zu nutzen. Während die Signatur die Ses-



siondaten schützt, würde die Nutzung von Pickle im Falle eines Leaks des geheimen Schlüssels nicht nur die Manipulation der Sessiondaten ermöglichen sondern auch das Ausführen beliebigen Codes.

Sofern der Funktionsumfang von cookiebasierten Sessions nicht ausreichen sollte, kann die Eigenschaft `session_interface` des Anwendungsobjekts auf eine benutzerdefinierte Klasse verweisen, die die Sessiondaten z.B. in Redis oder einer SQL-Datenbank abspeichert. Das Interface ist dabei sehr einfach gehalten; eine Redis-basierende Implementierung ist mit weniger als 70 Zeilen Code möglich.

Standardmäßig enthält Flask jedoch nur das Cookie-Interface, allerdings hat der Maintainer von Flask mit dem `RedisSessionInterface` ein entsprechendes Interface für serverseitige Sessions als Erweiterung veröffentlicht.

## Caching

Flask selbst enthält keinen Cache, allerdings bietet Werkzeug eine Lowlevel-Abstraktion verschiedener Cache-Backends, und stellt dabei die Methoden `get`, `set`, `delete` sowieso entsprechende `_many`-Varianten davon zur Verfügung. Dieses Cache-Interface ist oftmals ausreichend, allerdings bietet die Erweiterung *Flask-Cache* auch eine über die zentrale Flask-Konfiguration konfigurierbare High-Level-API dafür, die neben den Lowlevel-Funktionen Decorators zum Cachen ganzer Views und ein Jinja2-Plugin zum Cachen einzelner Templatefragmente.

Neben diesen relativ üblichen Cache-APIs bietet *Flask-Cache* einen `memoize`-Decorator. Dieser kann zum Cachen beliebiger Funktionen genutzt werden; er nutzt den Namen der Funktion und die übergebenen Parameter als Cache-Key und führt die eigentliche Funktion nur aus, wenn das Ergebnis noch nicht im Cache vorhanden ist.

## Integrierbarkeit

Flask benötigt weder eine bestimmte Datenbank noch setzt es eine bestimmte Verzeichnisstruktur in der Anwendung voraus. Während URL-Routingregeln meist über Decorators definiert werden, so ist die Nicht-Decorator-API genauso mächtig und ähnlich komfortabel. Daher ist es sehr einfach, Flask in eine existierende Anwendung zu integrieren.

Selbstverständlich kann eine größere Anwendung, in der Flask nachträglich integriert wurde, nicht alle Vorteile des Frameworks ausnutzen ohne dass der

gesamte Code modifiziert werden muss. Nichtsdestotrotz ist Flask sehr flexibel und so ist es kein Problem, bestehende Views funktionsfähig in Flask einzubinden und neuen Code im „Flask-Stil“ zu schreiben.

### Erweiterbarkeit

Flask bietet mit *Signals* eine API, um Callbacks zu registrieren, die zu verschiedenen Zeitpunkten ausgeführt werden. Diese sind unter anderem das Rendern eines Templates und vor bzw. nach einem Request. Sofern dies nicht ausreicht, kann die `Flask`-Klasse subclassed werden. Dies bietet die Möglichkeit, bestehende Funktionalität zu erweitern bzw. zu ersetzen und z.B. die `Request`-Klasse durch eine eigene Subklasse davon zu ersetzen.

Flask-Extensions nutzen in der Regel ausschließlich die Callbacks, sodass sie einfach zu einer Anwendung hinzugefügt werden können und unabhängig von anderen Extensions sind. Um eine Extension zu benutzen, sind zwei Methoden verbreitet. Für einfache Extensions, die keine eigene für den Entwickler relevante Klasse besitzen, wird empfohlen, eine Funktion `init_app(app)` zu definieren. Diese akzeptiert als ersten Parameter die Flask-Anwendung und registriert relevante Callbacks und führt sonstigen Initialisierungscode aus. Komplexere Extensions enthalten in der Regel eine Klasse, über die vom Entwickler auf die Extension zugegriffen werden kann. Dabei wird das Applikationsobjekt über einen optionalen Parameter an den Konstruktor übergeben; sofern er nicht genutzt wird, sollte die Klasse die zuvor erwähnte Methode `init_app(app)` besitzen. Der Zweck davon ist, die Nutzung von *application factories* zu ermöglichen, d.h. Funktionen, die die Applikation erst erstellen und damit möglicherweise nie global verfügbar machen. In diesem Fall ist es möglicherweise notwendig, die Instanz der Extensionklasse zu erstellen, obwohl noch keine Flask-Instanz existiert.

Sofern keine dieser Optionen mächtig genug sind, besteht auch die Möglichkeit, eine der Werkzeug-Klassen durch eine eigene Subklasse zu ersetzen um beispielsweise das URL-Routing-System direkt verändern zu können. Wie bei jedem Framework besteht darüber hinaus die Möglichkeit, das Framework zu forken und daraufhin den Frameworkcode selbst zu verändern.

### Sonstige Features

Flask nutzt Proxy-Objekte um `request`, `g` und `session` global verfügbar zu machen und trotzdem jeweils - auch bei mehreren Threads - auf das richtige Objekt zu verweisen. Dadurch müssen diese Objekte niemals als Funktionspa-

parameter übergeben werden, sondern können einfach aus dem `flask`-Package importiert werden.

Während `request` und `session` ziemlich selbsterklärend sind, handelt es sich bei `g` um ein von Flask selbst nicht genutztes Objekt, welches allerdings sowohl Anwendungscode als auch Flask-Erweiterungen eine Möglichkeit bietet, requestspezifische Daten abzulegen. Beispielsweise ist der aktuell eingeloggte User meist in `g.user` gespeichert und oftmals nutzen Datenbank-Extensions das Objekt, um die gerade aufgebaute Datenbankverbindung beim nächsten Zugriff nicht erneut aufbauen zu müssen.

Für viele Standardfeatures existieren bereits fertige Erweiterungen, sodass man diese nicht selbst implementieren muss. Neben komplexen Erweiterungen wie einem Django-ähnlichen dynamischen CRUD-Administrationsbereich, ORM-Systemen und einer kompletten Benutzerverwaltung samt Rechtesystem gibt es auch viele kleinere Extensions, die hauptsächlich dem Entwicklerkomfort dienen und z.B. eine Redis-Verbindung anhand der Daten in der Flask-Konfiguration aufbauen oder das Verschicken von E-Mails über SMTP mithilfe verschiedener Python-Libraries mit einer intuitiven API vereinfachen.

Flask nutzt den von Werkzeug als WSGI-Middleware bereitgestellten webbasierten Debugger. Während dieser nicht den Funktionsumfang eines vollwertigen Debuggers bietet - insbesondere Stepping und Breakpoints werden nicht unterstützt - so ist er bei der Entwicklung sehr hilfreich, um kleinere Fehler schnell zu finden und mithilfe der integrierten Python-Shell Code im selben Kontext auszuführen, in dem die Exception verursacht wurde.

## **Dokumentation**

Flask, Werkzeug und Jinja2 besitzen jeweils eine sehr ausführliche Online-Dokumentation, die sich jeweils auf die aktuellste Version bezieht. Features, die erst in einer bestimmten Version hinzugefügt wurde, sind entsprechend markiert, sodass die Dokumentation auch bei Nutzung einer älteren Version noch hilfreich ist. Darüber hinaus sind für jeden Versionssprung Updatehinweise verfügbar, die explizit auf potenziell problematische Änderungen hinweisen.

Ein großer Teil der ausführlichen API-Dokumentation ist anhand der im Code verwendeten Docstrings generiert. Dies hat den Vorteil, dass beim Lesen des Codes die relevante Dokumentation direkt sichtbar ist. Kommentare im Code sind wie im Python-Styleguide empfohlen an allen Stellen vorhanden, die nicht selbsterklärend sind.

## Lizenz

Sowohl Flask als auch Werkzeug und die übrigen zwingend notwendigen Python-Libraries stehen unter der BSD-Lizenz. Für Flask-Erweiterungen wird eine ähnlich freizügige Lizenz empfohlen bzw. im Falle von *approved extensions* sogar zwingend notwendig. Die offizielle Dokumentation empfiehlt entweder die BSD-Lizenz, die MIT-Lizenz oder die größtenteils dem amerikanischen Public-Domain-Konzept entsprechende WTFPL<sup>25</sup>.

---

<sup>25</sup>Do What the Fuck You Want to Public License

## 4. Auswahl eines Frameworks

### 4.1. Migrationspfade

Sowohl die einzelnen Frameworks als auch die Kombination eines dieser Frameworks mit Teilen des Indico-Frameworks haben spezifische Vor- und Nachteile. Im Folgenden werden sowohl der vollständige Umstieg auf ein bestimmtes Framework als auch Hybridlösungen vorgestellt und bewertet.

Der Vollständigkeit halber wird auch die - zugegeben sehr unwahrscheinliche - Option, komplett bei der bestehenden Lösung zu bleiben, kurz betrachtet.

#### 4.1.1. Weiternutzung des Indico-Frameworks

Getreu dem Motto „never change a running system“ stellt sich natürlich die Frage, ob sich der nicht zu unterschätzende Aufwand, eine komplexe Software wie Indico auf ein neues Framework umzustellen, überhaupt lohnt. Diese Frage ist im Fall von Indico aus verschiedenen Gründen zu bejahen. Insbesondere die Kombination aus einem sehr einfachen Routingsystem und dem URL-Dateisystem-Mapping im CGI-Stil bietet bei der Entwicklung neuer Features diverse Nachteile: Sofern das neue Feature eine für den Benutzer direkt zugängliche Seite besitzt ist aus Gründen der Einheitlichkeit eine *\*.py*-URL angebracht. Im Falle eines neuen Moduls wie der HTTP-API ist dies jedoch nicht gegeben, weshalb sich dort anbietet, ein neues Pfadsegment einzuführen und über dieses alle Anfragen in das entsprechende Modul zu routen. Dies bedeutet jedoch, dass jegliches weitere Routing von dem jeweiligen Modul übernommen werden muss.

Ein enormer Vorteil bei der bestehenden Lösung ist natürlich, dass sie seit Jahren relativ stabil läuft und von einer breiten Userbasis benutzt wird und damit die „Kinderkrankheiten“ eines neuen Systems nicht vorhanden sind. Ebenfalls ist die ZODB-Datenbank stark in das aktuelle System integriert, wobei anzumerken ist, dass dabei teilweise fast derselbe Code an mehreren Stellen verwendet wird: Die RH-Klassen und die JSON-RPC-API verwenden jeweils dasselbe Retry-System, um

Datenbankkonflikte zu behandeln, jedoch ist es in den jeweiligen Klassen unabhängig voneinander implementiert.

Ein Vorteil beim Verzicht auf ein neues Framework wäre natürlich auch, die dafür notwendige Entwicklungszeit anderweitig nutzen zu können. Dies ist jedoch nur kurzfristig gesehen ein Vorteil, da die Nichtnutzung eines modernen Frameworks in Zukunft bei neuen Features mit hoher Wahrscheinlichkeit zu deutlich mehr Entwicklungsaufwand führt als mit einem entsprechenden Framework notwendig wäre.

Insbesondere aufgrund dieser Folgekosten ist ein neues Framework also unbedingt notwendig. Daher wird die Option, weiterhin ausschließlich das Indico-Framework, in den folgenden Abschnitten nicht mehr betrachtet.

#### 4.1.2. Vollständige Migration zu Django

Aufgrund der Struktur von Django - mehrere möglichst in sich abgeschlossene Apps - und der empfohlenen Verzeichnisstruktur in den einzelnen Apps ist es schwierig, Django in die bestehende Indico-Codebasis zu integrieren. Es ist deutlich einfacher und auch sauberer, das Django-Projekt in einem neuen Verzeichnis unabhängig von der bestehenden Codebasis zu integrieren. Danach müssen zunächst die in der gesamten Anwendung genutzten Funktionen wie das Benutzer- und Gruppensystem samt LDAP-Anbindung unter Verwendung der Django-User-App neu implementiert werden. Obwohl das bestehende Benutzersystem nicht Teil des Frameworks ist und durchaus in das Django-Projekt übernommen werden könnte, hat die Neuimplementierung den Vorteil, dass die Django-Infrastruktur im weiteren Verlauf möglichst effizient benutzt werden kann. Da die Benutzer in der ZODB abgespeichert werden, muss diese Datenbank auch in Django integriert werden; dazu bietet sich evtl. *Django-ZODB* an - dies muss jedoch genauer untersucht werden, da unter anderem Datenbankkonflikte während eines Commits sauber abgefangen werden sollten. Eine Alternative wäre natürlich der Umstieg auf eine relationale Datenbank unter Verwendung des ORM-Systems von Django, allerdings würde dies die bereits sehr aufwändige Migration noch komplexer machen.

Sobald der Kern der Django-Version von Indico funktionsfähig ist, kann die eigentliche Anwendung migriert werden. Da Indico mit den RH-Klassen bereits klassenbasierte Views nutzt und oftmals auch Vererbung nutzt, bietet es sich in den meisten Fällen an, auch weiterhin Klassen statt einfacher Funktionen zu nutzen. Während die meiste Anwendungslogik dieser Klassen übernommen werden kann, sind dennoch

in fast allen Fällen Änderungen angebracht, um z.B. über die von Django bereitgestellten Variablen auf HTTP-Parameter und dynamische Elemente aus der URL zuzugreifen. Zusammen mit dem Umstieg auf die Django-Templateengine macht dies also die Anpassung hunderter Klassen und Templates notwendig. Letzteres kann teilweise automatisiert werden, die Klassen müssen jedoch manuell umgeschrieben werden.

Für das URL-Routing müssen entsprechende Regeln in Form regulärer Ausdrücke geschrieben werden. Indico besitzt ca. 700 verschiedene URLs, die zwar oftmals dieselben Parameter wie die ID des Events enthalten und damit dank der `include()`-Funktion des Routingsystems vereinfacht werden können, aber dennoch manuell in *clean URLs* umgeschrieben werden müssen - die Entscheidung, statt `/conferenceTimetable.py/pdf` die URL `/event/123/timetable.pdf` zu nutzen, kann keine Software übernehmen. Zusätzlich zu den neuen URLs müssen für alle direkt verlinkbaren Seiten auch die alten URLs weiterhin unterstützt werden, um Links von anderen Seiten oder Suchmaschinen nicht in einer Fehlermeldung enden zu lassen. Indico nutzt derzeit die `URLHandler`-Klassen zur URL-Generierung. Daher existiert eine zentrale Stelle, die angepasst werden muss, um die neuen URLs auch überall zu erzeugen, wo Indico eine URL generiert.

Zusammenfassend kann man sagen, dass es sich bei dieser Migration um ein extrem aufwändiges und zeitintensives Unterfangen handelt, das schon eher ein Rewrite als eine Migration ist - insbesondere wenn man es mit einem Wechsel von der bestehenden Objektdatenbank auf eine relationale Datenbank kombiniert. Am Ende führt der Umstieg jedoch zu sauberem Code, der sowohl leicht wartbar ist als auch für neue Entwickler einen einfachen Einstieg ermöglicht, da viele Python-Entwickler bereits mit Django vertraut sind.

### 4.1.3. Erweiterung durch Django

Obwohl es wie bereits erwähnt schwierig ist, Django in die bestehende Codebasis von Indico einzubauen, ist die Integration natürlich prinzipiell möglich, sofern man die von den Django-Entwicklern empfohlene Anwendungsstruktur ignoriert. In diesem Fall würde man kein Grundgerüst mit `django-admin.py` generieren, sondern die entsprechenden Django-Objekte bzw. -Funktionen direkt in bestehenden oder neuen Dateien innerhalb des Indico-Packages importieren. Dazu bietet sich ein neues Subpackage `indico.web.django` an, welches den gesamten Django-spezifischen Basiscode enthält. Der WSGI-Code des alten Indico-Frameworks in

`indico.web.wsgi` kann in jedem Fall entfernt werden, wobei dementsprechend natürlich auch das stark auf diesem Code basierende Sessionssystem entfernt und durch das von Django bereitgestellte System ersetzt werden muss. Danach können entweder die im *mod\_python*-Stil geschriebenen Funktionen über das Routingsystem von Django aufgerufen werden, oder aber die RH-Klassen entsprechend angepasst werden, um direkt von Django aufgerufen werden zu können. Durch die vollständige Weiterverwendung dieser Klassen würde der ZODB-Zugriff vom bestehenden Code übernommen werden. Für GET/POST-Daten und URL-Parameter muss eine entsprechende Kompatibilitätsschicht entwickelt werden, die sie in das bisher genutzte Format konvertiert. Durch die Weiterverwendung des Mako-Templatesystems kann ebenfalls viel Entwicklungsaufwand gespart werden, wobei eine externe Templateengine bei weitem nicht so gut mit dem Framework zusammenspielt wie die framework-eigene Engine.

Letztendlich lässt sich bei diesem Migrationspfad sehr viel Entwicklungszeit einsparen, allerdings stellt sich die Frage, welchen Nutzen man außer den schöneren URLs am Ende hat. Immerhin handelt es sich bei Django um ein sehr mächtiges und komplexes Framework, welches man bei dieser Lösung nur zu einem Bruchteil nutzen würde - und ein mit Django vertrauter Entwickler müsste sich dennoch erst einarbeiten, da fast keine der bei Django üblichen *Best Practices* genutzt werden könnten.

#### 4.1.4. Erweiterung durch Flask

Derzeit nutzt Indico den von Werkzeug bereitgestellten Python-Webserver um bei der Entwicklung auch ohne Apache und *mod\_wsgi* betrieben werden zu können. Daher bietet es sich an, das auf Werkzeug basierende Flask-Microframework in Indico zu integrieren. Neben der in einem späteren Abschnitt beschriebenen Komplettmigration ist insbesondere eine partielle Migration bzw. Erweiterung der bestehenden Codebasis auf Flask interessant. Zu Beginn bietet es sich an, den WSGI-Kern und darauf aufbauenden Code - insbesondere also das Sessionssystem - des Indico-Frameworks durch Flask zu ersetzen, jedoch die einwandfrei funktionierende Datenbankanbindung beizubehalten. Das Entfernen des WSGI-Kerns ist ein guter Ausgangspunkt, da dadurch sicher obsolet werdender Code entfernt wird. Nachdem dies geschehen und Flask integriert ist, ist zu prüfen, ob noch weitere Kernbestandteile von Indico von dem alten WSGI-Code abhängig sind und angepasst bzw. ersetzt werden müssen. Danach kann zunächst ein einfacher Wrapper das Routing im *mod\_python*-Stil



emulieren und somit Indico mithilfe des Flask-WSGI-Backends größtenteils lauffähig machen.

Die wichtigste von Flask bereitgestellte Neuerung ist aber natürlich die Kombination aus dem Routingsystem und der Modularität von Blueprints. Neben mit vertretbarem Aufwand integrierbaren Features wie dem `g`-Objekt und dem integrierten Debugger ist das URL-Routing die komplexeste Flask-basierte Erweiterung, und somit auch die aufwändigste. Allerdings ist auf *Rules* und *Endpoints* basierende Struktur sehr ähnlich mit der derzeit in Indico genutzten URL-Erzeugung. Daher besteht eine gute Chance, dass abgesehen vom Definieren der neuen URLs viele Aufgaben in diesem Bereich automatisiert werden können. Gerade im Hinblick auf die Kompatibilität mit Links auf die alten URLs ist dies sehr hilfreich.

Der Vorteil, Flask zu nutzen, aber nicht rigoros jedes von Flask oder einer Flask-Erweiterung gebotene Feature sofort einzuführen, liegt insbesondere in der relativ einfachen Migration. Neben der unabhängig vom Framework aufwändigen Umstellung der URLs müssen weder zwangsläufig Templates konvertiert noch alle View-Klassen angepasst werden. Gleichzeitig kann neuer Code aber durchaus zusätzliche Flask-Features nutzen, sodass auch nach der Migration die Codequalität von Indico langfristig immer besser wird, solange neue Features implementiert oder bestehende refactored werden.

#### 4.1.5. Vollständige Migration zu Flask

Bei einer Komplettmigration bietet die in Abschnitt 4.1.4 beschriebene Teilmigration einen guten Ausgangspunkt, da am Ende ein funktionierendes Indico steht, welches bereits an verschiedenen Stellen Flask-Features nutzt. Die verbleibenden Bereiche, insbesondere Templates und RH-Klassen, können zwar relativ unabhängig voneinander migriert werden, allerdings müssen in beiden Fällen diese Klassen bzw. ihre Nachfolger angepasst werden, weshalb es sinnvoll ist, diese schrittweise gemeinsam zu migrieren, d.h. jeweils eine Klasse und die von dieser genutzten Templates. Die Templatemigration kann größtenteils automatisiert werden, allerdings enthalten einige Templates Python-Code, der manuell entfernt bzw. in die dazugehörige Python-Klasse verschoben werden muss. Bei den weiteren Änderungen an den Klassen gibt es mehrere Möglichkeiten, Dinge wie die Datenbankverbindung und die dazugehörigen Transaktionen neu zu implementieren. Neben einem *Class Decorator* wäre auch eine Superklasse denkbar, die ähnlich wie derzeit die RH-Klasse die entsprechende Logik implementiert, aber dabei flexibel bleibt und Flask ideal nutzt.

Da es sich bei dieser Variante um eine Erweiterung der Einbindung von Flask in die bestehende Infrastruktur handelt, macht es wenig Sinn, diese zu schon Beginn in Betracht zu ziehen. Während sie logischerweise zu besserem Code führt, ist sie extrem aufwändig und bietet insbesondere im Hinblick auf neue Entwicklungen nur wenige Vorteile, da diese sowieso bereits die meisten neuen Features von Flask nutzen können.

## 4.2. Entscheidung für ein Framework

Nach der Analyse der diversen Frameworks und der verschiedenen Migrationspfade sind nun genügend Informationen vorhanden, um eine Entscheidung treffen zu können. Bei dieser muss allerdings auch bedacht werden, dass sie dauerhaft sein muss. Sobald die Migration abgeschlossen bzw. weit fortgeschritten ist, wäre es nicht wirtschaftlich, erneut die Technologie zu wechseln zumal damit auch die übrigen Entwickler dazu gezwungen würden, sich wieder mit einem neuen Framework auseinanderzusetzen.

Die Idee, das Indico-Framework vollständig zu ersetzen, ist zwar diskussionswürdig, kann aber gerade bei einem von einer Vielzahl an Benutzern aktiv verwendeten System, bei dem es häufig kleinere Bugreports, aber auch Feature-Requests gibt, nicht wirklich in die Tat umgesetzt werden, da ein Entwickler nicht langfristig ausschließlich an einer solchen Migration, die fast einer Neuentwicklung entspräche, arbeiten kann. Die große Zahl zu ändernder Klassen und Funktionen würde möglicherweise sogar die Mitarbeit weiterer Entwickler benötigen. Daher stellt sich nur die Frage, durch welches Framework Indico erweitert werden soll, welche Bereiche dabei unbedingt angepasst bzw. umgeschrieben werden müssen und wo es nur *nice to have* wäre, aber nicht unbedingt im Rahmen der eigentlichen Frameworkmigration geschehen muss.

Sowohl bei Django als auch bei Flask handelt es sich um stabile und verbreitete Webframeworks, hinter denen jeweils große Entwicklercommunities stehen und für die es viele, ebenfalls meist unter einer kompatiblen Open-Source-Lizenz stehende, Erweiterungen gibt. Der Unterschied liegt jedoch darin, dass Django-Erweiterungen oftmals vollständige *Apps* bereitstellen, die möglichst vollständig in eine eigene Anwendung integriert werden können und oftmals bereits eigene Templates mitliefern, während Flask-Extensions sich meist an Entwickler richten und Tools bzw. APIs bereitstellen. Sowohl diese Tatsache als auch der generelle Aufbau macht Django eher

bei neuen Projekten zum Framework der Wahl, da man dort keine Altlasten hat und damit alle Features des Frameworks nutzen kann, sofern sie für die Anwendung nützlich sind, statt gegen das Framework zu arbeiten um es mit bestehendem Code zu verbinden.

Beide Frameworks enthalten ein flexibles und mächtiges Routingsystem für URLs, welches auch URLs generieren kann. Die Designentscheidung in Django, dort durchgehend, d.h. auch beim Definieren der URLs, reguläre Ausdrücke zu verwenden, macht es jedoch nicht sehr entwicklerfreundlich und unnötig kompliziert. Im Gegensatz dazu bietet Flask einfache Platzhalter und optionale Konverter, die durch eine sehr einfache API erweitert werden können.

Aufgrund der partiellen Migration spielt die Templateengine der Frameworks keine große Rolle, da die bereits vorhandenen Mako-Templates nicht konvertiert werden. Unabhängig davon ist die Templatesyntax von Django und Jinja2 nahezu identisch. Dass Flask selbst keine Datenbank nutzt und daher auch kein ORM oder sonstige Datenbankschichten enthält ist aufgrund der in Indico genutzten ZODB von Vorteil; eine separate SQL-Datenbank für das Framework würde nur den Administrationsaufwand erhöhen.

Django und Flask nutzen beide die `dict`-API für Sessions. Dabei hat Django den Vorteil, bereits nativ einen In-Memory-Cache zum Speichern der Sessiondaten zu unterstützen, während Flask dazu ein separates Modul benötigt. Flask hingegen ist bei der Sessionklasse flexibler; zusätzlich zur `dict`-API lässt sich die Session durch Subclassing einfach durch *Properties* erweitern. Dabei handelt es sich um Attribute, die automatisch eine durch eine Getter- und Setterfunktion aufrufen und somit zusätzliche Logik enthalten können. Gerade bei Sessiondaten, die auf in der ZODB gespeicherte Elemente verweisen, ist dies nützlich, um nur bei Bedarf auf die Datenbank zuzugreifen.

Indico nutzt nur einen Lowlevel-Cache, der relativ neu ist und eine saubere API bereitstellt, die große Ähnlichkeit mit den Lowlevel-Cache-APIs von Django und Flask-Cache hat. Daher muss dieser nicht unbedingt durch einen vom Framework bereitgestellten Cache ersetzt werden. Insbesondere bei den Lowlevel-APIs bieten jedoch beide Frameworks nahezu identische Funktionalität und ähnliche Backends.

Weder die Lizenz noch der Dokumentationsumfang bieten eine gute Basis um die Entscheidung für eines der Frameworks zu beeinflussen.

Letztendlich kann Django in einer Anwendung wie Indico seine Vorteile jedoch nicht

ausspielen und würde durch das verglichen mit Flask kompliziertere Routingssystem den Aufwand während der Migration erhöhen. Im Gegensatz dazu ist Flask als Microframework, das keine bestimmten Technologien voraussetzt, sehr einfach integrierbar.

**Daher ist die Entscheidung zugunsten von *Flask* erfolgt.**

## 5. Migration zu Flask

Bei der Migration zu Flask sind einige Schritte am Anfang zwingend notwendig, während andere optional sind und in relativ beliebiger Reihenfolge ausgeführt werden können. Im Folgenden werden die Migrationsschritte in der Reihenfolge beschrieben, wie sie durchgeführt wurden.

### 5.1. Vorbereitung

In der Vorbereitungsphase wird Flask eingebunden und lauffähig konfiguriert. Sollten dabei jedoch Konflikte auftreten, werden diese bereits behoben. Hierzu können u.U. Features des Frameworks benutzt werden. Es findet zu diesem Zeitpunkt jedoch noch keine Migration statt. Das Endziel dieser Phase ist ein temporärer Wrapper, der die *mod\_python*-Funktionen über Flask aufruft, sodass grundsätzliche Inkompatibilitäten zwischen Flask und dem Indico-Code zu Beginn der nächsten Phase beseitigt werden können.

#### 5.1.1. Installieren von Flask

Um einem Python-Projekt eine neue Abhängigkeit hinzuzufügen gibt es grundsätzlich zwei Möglichkeiten. Insbesondere für Python-Libraries und andere über PyPi<sup>26</sup> veröffentlichte Packages nutzt man eine Datei *setup.py*, in der unter anderem die Funktion `setup()` aufgerufen wird, an die neben diversen anderen Metadaten über den Parameter `install_requires` die zur Installation benötigten Pakete übergeben werden. Dies ermöglicht es Paketmanagern, die notwendigen Abhängigkeiten automatisch zu installieren. Die Alternative zur *setup.py* ist die *requirements.txt*-Datei. Diese enthält jeweils ein Paket pro Zeile, wobei neben dem Paketnamen auch eine Version oder ein Verweis auf ein Versionskontrollsystem wie Git angegeben werden kann. Meist wird die *requirements.txt* bei Python-Anwendungen benutzt, die nicht über PyPi installiert sondern manuell heruntergeladen werden müssen und weitere Konfiguration benötigen. In diesem Fall enthalten die Installationsanweisungen

---

<sup>26</sup>Python Package Index, <https://pypi.python.org>

meist den Hinweis, die Abhängigkeiten mit den Befehl `pip install -r requirements.txt` zu installieren.

Da bei Indico ursprünglich vorgesehen war, es systemweit zu installieren und nur die Konfigurationsdaten und sonstige dynamische Dateien bzw. Verzeichnisse außerhalb des systemweiten Python-Verzeichnisses abzulegen, besitzt es sowohl eine `setup.py` als auch eine `requirements.txt`. Daher muss Flask an beiden Stellen als Abhängigkeit definiert werden und danach installiert werden. Dies kann entweder manuell mittels `pip install Flask` geschehen oder wie zuvor erwähnt über die `requirements.txt`. Letzteres hat den Vorteil, dass Tippfehler o.ä. direkt auffallen.

---

```
1 git+https://github.com/miracle2k/webassets.git
2 Werkzeug==0.9
3 Flask==0.10
```

---

**Listing 14:** Auszug aus der `requirements.txt` von Indico

Listing 14 zeigt einen Ausschnitt aus der `requirements.txt`, wobei unter anderem Flask und das zugrundeliegende Werkzeug-Toolkit in der gerade aktuellen Version eingebunden werden. Flask selbst setzt zwar bereits `'Werkzeug>=0.7'` voraus, allerdings erlaubt dies dem Paketmanager, eine beliebige Version ab 0.7 zu installieren. Meist ist dies kein Problem, da gute Libraries API-Inkompatibilitäten möglichst vermeiden. Allerdings ist es sicherer, dennoch von allen genutzten Abhängigkeiten die Version festzulegen, sodass ein Benutzer exakt dieselben Versionen nutzt, mit denen die Anwendung auch getestet wurde.

### 5.1.2. Einbinden von Flask

Indico enthält zwei Python-Packages: `MaKaC` und `indico`, wobei `MaKaC` alten Legacy-Code und `indico` hauptsächlich neuen Code enthält. Web-spezifischer Code ist im Paket `indico.web`, weshalb es sich anbietet, für Flask ein neues Package `indico.web.flask` anzulegen. Um den Flask-Code innerhalb dieses Pakets bereits im Hinblick auf die Migration und zukünftige Entwicklungen zu strukturieren, wird ein Modul `app.py` angelegt, welches die *Application Factory* und den zum Initialisieren der App notwendigen Code enthält.

Bei der *Application Factory* handelt es sich um eine Funktion, die eine Flask-Instanz erstellt, konfiguriert und zurückgibt. Diese Kapselung hat den Vorteil, dass jegliche

Initialisierung an einigen wenigen Stellen stattfindet und z.B. für Unittests eine Instanz mit anderen Konfigurationsoptionen erstellt werden kann. Ohne Nutzung einer Factory-Funktion würde die Flask-Instanz zur Importzeit des Moduls erstellt werden und jedes andere Modul könnte das Objekt von dort importieren und verändern. Zur „Laufzeit“, also während der Abarbeitung eines Requests, macht sich der Unterschied nicht mehr bemerkbar, da Flask mit `current_app` ein Proxy-Objekt bereitstellt, welches jeweils auf die gerade aktive Flask-Instanz verweist.

Derzeit wird der in Indico integrierte Webserver mittels `indico_shell -web-server` gestartet. Zu Beginn ist es sinnvoll, über einen weiteren Kommandozeilenparameter zwischen Flask und dem bestehenden Webserver samt dem alten Framework wählen zu können. Aufgrund des einfachen Aufbaus von WSGI muss dazu nur entweder das bestehende *application-Callable* oder das neue, von `make_app()` erstellte, an den Webserver übergeben werden. Beim Aktualisieren des dafür zuständigen Codes fällt auf, dass dort eine einfache WSGI-Middleware implementiert wurde, die die WSGI-Applikation über den Indico-Konfiguration angegebenen Pfad verfügbar macht - also `http://localhost:8000/indico` statt nur `http://localhost:8000`. Ein Blick in die Werkzeug-Dokumentation zeigt, dass dort mit `DispatcherMiddleware` eine fertige Middleware enthalten ist, die diese Aufgabe ebenfalls übernehmen kann - allerdings mit weniger benutzerdefiniertem Code. Da es sich dabei um eine sehr einfache Änderung handelt und Werkzeug sowieso bereits eingebunden war, bietet es sich an, diese bereits zu diesem Zeitpunkt vorzunehmen. Dazu dient die in Listing 15 gezeigte Funktion.

---

```
1 def make_indico_dispatcher(wsgi_app):
2     baseURL = Config.getInstance().getBaseURL()
3     path = urlparse.urlparse(baseURL)[2].rstrip('/')
4     if not path:
5         return wsgi_app
6     else:
7         return DispatcherMiddleware(NotFound(), {
8             path: wsgi_app
9         })
```

---

**Listing 15:** Indico-WSGI-Dispatcher

Sie akzeptiert als Parameter eine WSGI-Anwendung - also entweder die Flask-Instanz oder die alte Indico-WSGI-Applikation und gibt eine Anwendung zurück,

die die übergebene Anwendung in dem konfigurierten Pfad verfügbar macht und für alle anderen Pfade einen *404 Not Found*-Fehler an den Client sendet. Sofern kein Pfad konfiguriert ist gibt die Funktion direkt die App zurück, da die Middleware in diesem Fall nicht benötigt wird.

An dieser Stelle enthält Indico jetzt bereits eine lauffähige Flask-Anwendung, der man bereits zum ersten Testen eine Hello-World-Funktion hinzufügen könnte. Allerdings kann die Anwendung ausschließlich über den nicht für Produktionszwecke geeigneten Entwicklungswebserver gestartet werden. Um sie auch über einen WSGI-kompatiblen externen Webserver nutzen zu können, wird eine *indico.wsgi*-Datei benötigt, die die Anwendung unter dem Namen *application* zugänglich macht.

---

```
1 from indico.web.flask.app import make_app
2 application = make_app()
```

---

**Listing 16:** *indico.wsgi*

### 5.1.3. Interface zu den *mod\_python*-Funktionen

Beim *mod\_python*-Routing verweisen URLs der Form */file.py* bzw. */file.py/func* auf eine gleichnamige Datei und entweder die angegebene Funktion oder auf eine Funktion namens *index*. Der erste Parameter ist jeweils das Request-Objekt, danach folgen alle GET- und POST-Parameter als Keyword Arguments. Der Rückgabewert der Funktion wird als Antwort-Body an den Client gesendet.

Um ein Interface zu diesen Funktionen zu entwickeln, existieren verschiedene Möglichkeiten, die jedoch alle auf einer Art Reflection basieren. Neben komplexen, auf dem Python-Syntaxbaum aufbauenden, Varianten, ist die einfachste Lösung, die Dateien zu importieren und dann aus ihrem globalen Namespace alle aufrufbaren Objekte auszufiltern. Dies ist in einer dynamischen Sprache wie Python sehr einfach möglich: Mit `execfile(path, globals)` bietet Python eine Funktion, die die angegebene Datei ausführt und dabei das als *globals* übergebene *dict* für alle *globals*, d.h. die im aktuellen Modul verfügbaren globalen Variablen, nutzt. Nach dem Aufruf dieser Funktion muss der Inhalt der Liste also nur noch gefiltert werden, um nicht ausführbare Objekte zu entfernen, und enthält danach nur noch die relevanten Funktionen. Aus Dateiname und Funktionsname kann nun mithilfe einfacher Stringoperationen eine entsprechende Routingregel für Flask erzeugt werden.



Anders als *mod\_python* übergibt Flask jedoch keine Parameter an die Viewfunktionen, sofern die Routingregel keine solchen enthält. Daher ist eine Wrapperfunktion notwendig, welche die eigentliche Funktion mit den richtigen Parametern aufruft. Da es sich bei Funktionen in Python um First-Class-Objekte handelt, die ja wie in Abschnitt 2.1.6 erwähnt zur Laufzeit erstellt und von anderen Funktionen zurückgegeben werden können, ist dies wie man in Listing 17 erkennen kann sehr einfach möglich.

---

```
1 def create_flask_mp_wrapper(func):
2     def wrapper():
3         args = request.args.copy()
4         args.update(request.form)
5         flat_args = {}
6         for key, item in args.iterlists():
7             flat_args[key] = map(_to_utf8, item) if
                        len(item) > 1 else _to_utf8(item[0])
8         return func(None, **flat_args)
9     return wrapper
```

---

**Listing 17:** *mod\_python*-Wrapper-Factory

Beim Aufruf dieser Funktion mit der *mod\_python*-Funktion als Parameter wird eine neue Funktion definiert und zurückgegeben, die die GET- und POST-Parameter in einem einzigen Objekt zusammenfasst und alle Werte in UTF8 konvertiert. Dies ist nötig, da Flask Unicodestrings verwendet, während Indico intern grundsätzlich mit UTF8-Strings arbeitet. Neben None, dem Python-Äquivalent von dem aus anderen Sprachen bekannten NULL, statt des nicht mehr verfügbaren Request-Objekts übergibt die neue Funktion zum Schluss die neue Parameterliste in der von Indico erwarteten Form an die Originalfunktion.

## 5.2. Migration

### 5.2.1. Inkompatibilitäten beheben

Da statt eines (emulierten) *mod\_python*-Request-Objekts nun None übergeben wird, schlagen Zugriffe auf Methoden wie `req.get_remote_ip()` fehl. Flask macht

die IP des Benutzers über `request.remote_addr` zwar ebenfalls verfügbar, allerdings handelt es sich dabei grundsätzlich um die IP, von der aus die Verbindung zum Webserver aufgebaut wurde. Dies ist zwar bei der Entwicklung und in einfachen Setups die richtige IP-Adresse, allerdings nutzt z.B. die Indico-Installation am CERN einen Loadbalancer, der die IP-Adresse des Clients über den HTTP-Header *X-Forwarded-For* übergibt. Flask macht alle in diesem Header enthaltenen IP-Adressen über `request.access_route` zugänglich. Da die IP-Adresse des Loadbalancers jedoch nie relevant ist und der Header, sofern kein Loadbalancer verwendet wird, vom Benutzer manipuliert sein kann, ist eine neue Eigenschaft sinnvoll, die jeweils die korrekte IP-Adresse zurückgibt. Natürlich könnte dies einfach gelöst werden, indem man die bereits in Indico enthaltene Funktion `_get_remote_ip(req)` anpasst, d.h. den Parameter entfernt und auf die Flask-APIs zugreift. Flask ist jedoch flexibel genug, um eine deutlich sauberere Lösung zu implementieren. Durch Subclassing der Klasse `flask.wrappers.Request` kann die Funktion, die von der `remote_addr`-Eigenschaft genutzt wird, durch eine eigene ersetzt werden, die sofern ein Proxy vorhanden ist `self.access_route[0]` und ansonsten die `remote_addr`-Eigenschaft der Superklasse nutzt. Um die benutzerdefinierte `IndicoRequest`-Klasse in Flask auch zu verwenden, wird einfach `Flask` subclassed und die `request_class`-Klasseneigenschaft überschrieben. Alternativ könnte es auch erst in der durch `make_app()` erstellten Instanz überschrieben werden.

Ebenfalls mit Flask inkompatibel ist das Sessionssystem. Dies war aufgrund der starken Integration mit der *mod\_python*-Emulation zu erwarten und es sollte sowieso durch eine auf Flask basierende Neuentwicklung ersetzt werden. Da Flask standardmäßig nur auf signierten Cookies basierende Sessions unterstützt, ist die dort speicherbare Datenmenge auf ca. 4096 Bytes beschränkt. Ebenfalls treten insbesondere im Zusammenhang mit den in der JSON-RPC-API massiv genutzten AJAX-Requests, oftmals Probleme auf. Daher muss ein neues Sessioninterface entwickelt werden. Aufgrund des in Indico bereits vorhandenen Cache-Systems bietet es sich an, dieses zum Speichern der Sessiondaten zu verwenden. Gerade bei der empfohlenen Nutzung von Memcached oder Redis als Cache-Backend hat dies den Vorteil, dass für jeden Datensatz eine Time-To-Live angegeben werden kann und er nach Ablauf dieser Zeit automatisch gelöscht wird. In Flask eingebunden wird das Sessioninterface ähnlich wie die benutzerdefinierte `IndicoRequest`-Klasse: Die Klasse `Flask` enthält eine Eigenschaft, der man einfach eine Instanz der neuen Interfaceklasse zuweist. Das von dieser Klasse zu implementierende Interface ist dabei sehr simpel. Zu Beginn eines Requests wird die Methode `open_session(app, request)` auf-

gerufen und gibt das Sessionobjekt zurück, welches im Anwendungscode dann über den `session`-Proxy verfügbar ist. Nach erfolgreicher Verarbeitung des Requests kann die Methode `save_session(app, session, response)` die Sessiondaten speichern. Aus Performancegründen sollte die Session nur aktualisiert werden, wenn sie noch nie gespeichert wurde und Daten enthält, verändert wurde oder die Daten bzw. das Cookie bald ablaufen.

Das Sessioninterface ist jedoch ausschließlich für das Laden und Speichern der Session verantwortlich. Es enthält selbst keinerlei Sessiondaten. Diese sind in einer separaten Klasse, welche die Dict-API von Python implementiert und das Objekt nach jeder Änderung als *dirty* markiert. Die ursprüngliche Sessionimplementierung unterstützt neben einer Dict-ähnlichen API auch diverse Funktionen wie `getUser()`, die den aktuellen Benutzer zurückgeben. Da die neuen Sessions nicht mehr in der ZODB gespeichert werden, kann dort nicht direkt das Benutzerobjekt referenziert werden sondern ausschließlich die ID des Benutzers gespeichert werden. Dies macht es allerdings unmöglich, die Dict-API ohne größeren Aufwand dafür zu nutzen. Stattdessen werden diese speziellen Eigenschaften als *Properties* implementiert, also Objekteigenschaften, die intern auf eine Getter- und Setterfunktion zugreifen. Dies ermöglicht es, beim Zugriff auf `session.user` den entsprechenden Benutzer anhand der in der Session gespeicherten ID aus der Datenbank zu laden, für weitere Zugriff lokal zu cachen, und dann zurückzugeben.

Um das neue Sessionsystem zu nutzen, wird die alte Implementierung vollständig entfernt. Dies ist problemlos möglich, da `MaKaC.webinterface.session` keinen weiteren Code enthält und somit komplett gelöscht werden kann. Danach müssen alle darauf verweisenden Importe und natürlich der Code, der die alte Session initialisiert und im RH-Objekt ablegt, entfernt werden. Zugriffe auf explizit implementierte Funktionen der Session - also `getUser()` und ähnliche Methoden - lassen sich relativ leicht mit der Suchfunktion des Editors finden und auch fast automatisch durch ihre neue Variante in der Form `session.user` ersetzen. Nur Importe und Funktionsparameter müssen manuell angepasst bzw. entfernt werden. Sessionzugriffe über `getVar()` bzw. `setVar()` könnten zwar prinzipiell mit regulären Ausdrücken in die Dict-Syntax umgeschrieben werden, jedoch könnten in diesem Fall weder direkt in der Session abgelegte Datenbankobjekte noch die direkte Veränderung von bereits in der Session vorhandenen Daten gesondert behandelt werden. Ersteres ist jedoch prinzipiell nicht möglich und führt zu einem Fehler beim Laden der Session und letzteres würde die geänderten Daten nicht im Cache-Backend aktualisieren, da ein Container nicht ohne massive Performanceeinbußen feststellen kann, ob ein

enthaltenes Objekt verändert wurde. Daher werden alle weiteren Sessionzugriffe manuell angepasst. Dies geschieht allerdings noch nicht zu diesem Zeitpunkt, da es für einen allgemeinen Funktionstest von Indico nicht notwendig ist, dass jedes Feature fehlerfrei funktioniert.

Erste Tests zeigen an dieser Stelle, dass Indico nun schon prinzipiell auf Flask-Basis funktioniert. Allerdings schlagen alle Requests fehl, die eine im Dateisystem existierende Datei an den Client senden sollen, da dort *mod\_python*-spezifische Funktionen genutzt werden statt der komfortablen `send_file()`-Funktion von Flask. Diese Stellen lassen sich jedoch leicht finden und anpassen.

### 5.2.2. Trial and Error

An dieser Stelle ist es sinnvoll, alle einfach zugänglichen Bereiche von Indico kurz zu testen um weitere Probleme zu finden und zu beheben. Idealerweise könnte dies automatisiert durch die Unittests und funktionalen Tests geschehen, allerdings decken die Unittests hauptsächlich intern genutzten, nicht-web-spezifischen Code ab. Kombiniert mit sehr wenigen funktionalen Tests stellt sich schnell heraus, dass die vorhandenen Tests hier nicht wirklich hilfreich sind. Daher existieren zwei Optionen: Manuell testen oder entsprechend viele Testcases schreiben, um die ganze Software abzudecken. Dass letzteres langfristig gesehen die bessere Lösung ist steht außer Frage, allerdings muss hier aus Zeitgründen auf manuelle Tests zurückgegriffen werden.

Die Tests zeigen, dass unter anderem die HTTP-API und JSON-RPC-API nicht funktionieren, da sie das bei der Beschreibung des Indico-Frameworks erwähnte auf dem ersten Pfadsegment basierende Routing nutzen und dementsprechend vom zuvor entwickelten Wrapper nicht abgedeckt werden. Dies kann jedoch durch manuelles Hinzufügen der entsprechenden Routingregeln und geringfügige Modifikationen im davon aufgerufenen Code leicht behoben werden.

Ebenfalls in dieser Phase behoben werden Zugriffe auf das alte, nicht mehr vorhandene, Sessionobjekt. Dabei stellt sich heraus, dass das Room-Booking-Plugin extrem viele separate Einträge in der Session abspeichert, die sinnvoller in einem Dict zusammengefasst werden könnten. Da dieses Refactoring den Aufwand nur minimal erhöht wird es zusammen mit den sowieso notwendigen Änderungen durchgeführt.

Das Vorgehen nach dem *Trial-and-Error*-Prinzip hat hier den Vorteil, dass jeweils nur einzelne zusammengehörende Bereiche von Indico aktualisiert werden und sie direkt nach der Änderung auf Funktionalität getestet werden können.

### 5.2.3. Routing

In Abschnitt 5.1.3 wurde ein einfaches Interface zu den *mod\_python*-Funktionen entwickelt, um möglichst schnell ein lauffähiges System zu haben. Im Hinblick auf saubere URLs und Kompatibilität mit alten URLs ist es jedoch sinnvoller, die Routingregeln nicht erst zur Laufzeit zu generieren, damit die alten Python-Dateien, die alle exakt dieselbe Struktur wie das Beispiel in Listing 18 haben, gelöscht werden können.

---

```
1 from MaKaC.webinterface.rh import about as rhAbout
2
3 def index(req, **params):
4     return rhAbout.RHAbout(req).process(params)
```

---

**Listing 18:** about.py im htdocs-Verzeichnis

#### 5.2.3.1. Generieren der Routingregeln

Um die benötigten Daten aus den Python-Dateien zu extrahieren, existieren mehrere Möglichkeiten. Mittels regulärer Ausdrücke ließen sich sowohl die Importe als auch die Funktions- und Klassennamen relativ einfach extrahieren. Allerdings müssten Sonderfälle wie der im Beispiel genutzte Import-Alias entsprechend behandelt werden und auch unterschiedliche Formatierung entweder zuvor vereinheitlicht oder durch einen komplexeren regulären Ausdruck abgedeckt werden. Da es sich bei Python um eine extrem dynamische Sprache handelt, kann jedoch auch der *Abstract Syntax Tree* der jeweiligen Funktion genutzt werden, um an die benötigten Daten zu gelangen. Dieser Syntaxbaum enthält eine abstrakte Repräsentation des Quellcodes und ist sehr viel einfacher zu parsen als der nur in Stringform vorliegende Quellcode selbst.

Relevant sind um beim Beispiel aus Listing 18 zu bleiben neben dem bereits bekannten Funktionsnamen `index`, der zur Sicherheit mit dem Namen in *FunctionDef* verglichen werden kann, der globale Name `rhAbout` des Moduls und der Klassenname `RHAbout`. Diese beiden Werte sind wie man in der Abbildung 3 leicht erkennt vom inneren *Call* aus über `func.value.id` und `func.attr` erreichbar. Über die Globals des mittels `execfile()` ausgeführten Python-Moduls kann das zu `rhAbout` gehörende Python-Modul referenziert und der korrekte Name ausgelesen werden.



hinzuzufügen oder zu entfernen. In einem String-Kontext genutzt gibt das Objekt mithilfe der magischen Methode `__str__` die auf der `_relativeURL` basierende absolute URL zurück.

---

```
1 class UHAbout(URLHandler):  
2     _relativeURL = 'about.py'
```

---

**Listing 19:** URLHandler für `about.py`

Man könnte nun denken, dass es sehr einfach ist, die `URLHandler`-Klasse anzupassen, sodass statt der relativen URL der Flask-Endpoint übergeben wird und die URL-Instanz die mittels `url_for()` zum Endpoint gehörende URL erhält. Mit den bestehenden Routingregeln funktioniert dies auch ohne Schwierigkeiten. Problematisch wird es allerdings, sobald eine Regel eine Variable in der URL enthält. In diesem Fall schlägt `url_for()` fehl, sofern die Variable nicht übergeben wurde. Da die in Indico jedoch nicht immer alle Variablen an `getURL()` übergeben werden sondern teilweise erst über die Methoden der URL-Klasse hinzugefügt werden, darf `url_for()` erst aufgerufen werden, wenn die URL auch als String benötigt wird. Dies lässt sich dadurch lösen, dass der `URLHandler` den Endpoint und die bereits bekannten Parameter an die URL-Klasse weitergibt, und diese die URL erst dann generiert, wenn sie als String benötigt wird. Der Nachteil bei dieser Lösung ist, dass Tracebacks im Fehlerfall nicht sehr hilfreich sind: Oftmals wird die URL erst innerhalb eines Templates in einen String konvertiert und die Mako-Templateengine zeigt in Tracebacks nicht den im Template vorhandenen Code an sondern den aus dem Template generierten Python-Code.

Ein weiteres Problem zeigt sich in clientseitigem JavaScript-Code, der URLs generieren muss. Zuvor konnten dort einfach Parameter mittels Stringoperationen angehängt werden. Dies ist jedoch nicht mehr möglich, da die URL ohne alle notwendigen Parameter überhaupt nicht generiert werden kann. Die Lösung dazu liegt im Package `werkzeug.contrib.jsrouting`. Dabei handelt es sich um eine JavaScript-Implementierung des URL-Generators von Flask/Werkzeug. Während diese nicht direkt in Indico nutzbar ist, da eine Liste mit allen ca. 700 Routingregeln und später nochmal ebenso vielen Kompatibilitätsregeln die an den Browser gesendeten Daten unnötig aufblähen würde, bietet es eine sehr gute Basis für eine vereinfachte Version, die nicht die komplette Routingtabelle clientseitig verfügbar macht, sondern nur einige ausgewählte Regeln. Implementiert ist dies mit einer Eigenschaft der URL-Klasse, die die notwendigen Informationen der Regel als JSON zurückgibt und einer

auf dem jsrouting-Code basierenden JavaScript-Funktion, die aus der JSON-Regel und den Parametern die URL generiert.

Um die URL-Generierung von Flask nun tatsächlich zu verwenden müssen alle URLHandler-Subklassen entsprechend angepasst werden. Dies kann praktischerweise mit einem regulären Ausdruck automatisiert werden. Da die generierten Legacy-Routingregeln alle in einem Blueprint namens *legacy* enthalten sind, müssen auch die Endpoints in den URLHandler-Klassen auf diesen Blueprint verweisen.

---

```
1 class UHAbout(URLHandler):  
2     _endpoint = 'legacy.about'
```

---

**Listing 20:** Endpointbasierter URLHandler für about.py

#### 5.2.3.3. Clean URLs

Eines der Hauptziele der Migration, neben saubererem Code und dem damit verbundenen Komfort für Entwickler, ist die Nutzung sauberer URLs. Vor der Implementation macht es Sinn, sich Gedanken über den Aufbau dieser URLs zu machen. Ein oftmals in diesem Kontext genanntes Paradigma ist REST<sup>27</sup>. Dabei handelt es sich um die oftmals fälschlich als Standard bezeichnete Idee, dass eine URL genau eine Ressource bzw. Collection repräsentiert und verschiedene HTTP-Methoden unterschiedliche Aktionen darauf ausführen. Beispiel würde */event/* die Liste aller Events repräsentieren während */event/123* das Event mit der ID 123 repräsentieren würde. Insbesondere aufgrund der Verwendung verschiedener HTTP-Methoden wie DELETE und PUT eignet sich REST eher für APIs als für den von Benutzern direkt zugänglichen Bereich einer Webanwendung. Außerdem wäre es bei einer Anwendung wie Indico sehr umständlich, alle Ressourcen im REST-Stil zu repräsentieren. Deutlich einfacher und benutzerfreundlicher ist es, die URL-Struktur nach Kategorien aufzuteilen, sodass */event/123/* auf die Startseite des Events verweist, */event/123/timetable* auf den Zeitplan und */event/123/timetable.pdf* auf die PDF-Version von selbigem. Der Administrationsbereich der Anwendung ist komplett unabhängig davon und somit über */admin/* erreichbar.

Die eigentliche Implementierung der sauberen URLs ist prinzipiell einfach, aber sehr zeitaufwändig, da jede URL manuell definiert werden muss. Die generierten Legacy-

---

<sup>27</sup>Representational state transfer



Routen vereinfachen die Aufgabe jedoch etwas, da nach dem Erstellen der Flask-Blueprints nur die entsprechenden Legacy-Routen via Cut&Paste im entsprechend Modul eingefügt und die URL und die erlaubten HTTP-Methoden angepasst werden. Da die Endpoints denselben Namen wie im Legacy-Blueprint haben muss in den dazugehörigen URLHandler-Klassen nur der Name des Blueprints angepasst werden.

#### 5.2.3.4. Unterstützung der alten URLs

Für bestehende Indico-Installationen wäre es extrem problematisch, wenn nach dem Update auf eine neuere Indico-Version alle alten URLs in einer *404 Not Found*-Fehlermeldung enden würden. Daher müssen die alten URLs ebenfalls unterstützt werden; allerdings sollten sie nicht auf die ursprüngliche Zielseite verweisen, sondern mit einem *301 Moved Permanently*-Redirect auf die neue URL weiterleiten.

Aufgrund der in Abschnitt 5.2.3.1 gewählten Endpoint-Namen sind keine weiteren Informationen notwendig, um aus diesem Namen die ursprüngliche URL zu generieren. Daher wird einfach eine Funktion implementiert, die zu einem existierenden Blueprint einen Kompatibilitäts-Blueprint generiert, der die alten URLs nutzt und auf die neuen weiterleitet. Der einzige zu beachtende Sonderfall ist bei POST-Requests. Diese werden laut HTTP-Standard bei einer Weiterleitung mit GET wiederholt, was nicht gewollt ist. Daher muss in diesem Fall auf die Weiterleitung verzichtet werden und die Originalfunktion direkt aufgerufen werden. Da normale Links und Suchmaschinen jedoch ausschließlich GET verwenden, ist dies kein Problem.

### 5.3. Probleme bei der Migration

Die Migration zu Flask lief größtenteils unproblematisch ab, wobei gerade bei einer Anwendung dieser Größe immer einige kleinere Schwierigkeiten auftreten. Diese beruhen jedoch meist auf unerwarteten Sonderfällen wie alten *mod\_python*-Dateien, die Debugcode enthielten und somit beim Import unerwartete Nebeneffekte hatten.

Ein weiterer ursprünglich nicht bedachter Sonderfall machte sich beim Testen von Indico mit dem *nginx*-Webserver bemerkbar: Bei statischen Dateien, die im Dateisystem existieren, ist es deutlich effizienter, wenn der Webserver die Datei ausliefern kann, als wenn die Webanwendung dies tun muss. Dazu unterstützen die meisten Webserver den *X-Sendfile*-Header, der den Pfad zu der Datei enthält, die der Webserver an den Client senden soll. Aus Sicherheitsgründen unterstützt nginx jedoch nur

*X-Accel-Redirect*, der sich zwar fast identisch verhält, allerdings statt eines Dateisystempfads eine URL erwartet, die in der Webserverkonfiguration dann auf einen entsprechenden Pfad gemappt wird. Dank des Middleware-Systems von WSGI ließ sich dieses Problem jedoch leicht lösen, indem eine neue Middleware den *X-Sendfile*-Header entfernt und mithilfe einer zuvor konfigurierten Mappingtabelle den Pfad in eine URL konvertiert und diese im *X-Accel-Redirect*-Header weiterreicht.

## 5.4. Ausblick

Bei der Entwicklung neuer Features kann Flask deutlich stärker genutzt werden als es beim bestehenden Code der Fall ist. Insbesondere bei der Generierung von URLs macht es Sinn, direkt auf `url_for()` zuzugreifen, statt den Umweg über eine neue URLHandler-Subklasse zu gehen. Darüber hinaus kann strenger darauf geachtet werden, dass GET und POST korrekt eingesetzt werden. Ein normaler Seitenaufruf, der keinerlei Daten verändert, sollte grundsätzlich GET nutzen während die Veränderung von Daten ausschließlich mit POST möglich sein sollte. Dies kann durch das Routingsystem von Indico einfach durchgesetzt werden, allerdings hätte es im bestehenden Code oftmals Änderungen an den HTML-Templates benötigt, da insbesondere im Administrationsbereich häufig Formularbuttons zur Navigation verwendet werden. Für Fälle, in denen sowohl GET als auch POST benötigt werden, also hauptsächlich auf Seiten, die Formulare enthalten, ist es jedoch aufgrund der Architektur von Indico sinnvoller, dies innerhalb der RH-Klasse zu tun, da ansonsten im Anwendungscode manuell mittels `request.method == 'POST'` unterschieden werden müsste, welcher Codezweig gerade ausgeführt werden soll. Die Lösung dafür findet sich in der Klasse `MethodView` von Flask. Dabei handelt es sich um eine Klasse, die eine Methode mit demselben Namen wie das verwendete HTTP-Verb aufruft. Im Kontext von Indicos RH-Klassen würde also entweder `_process_GET()` oder `_process_POST()` aufgerufen.

Mittelfristig könnte man auch nach einer Alternative zu den RH-Klassen zu suchen, um in neuem Code weniger Zwischenschichten zu haben. Voraussetzung dafür ist aber, dass alle derzeit von der RH-Klasse übernommenen Aufgaben anderweitig gelöst werden können. Insbesondere die oftmals überschriebene Methode zum Überprüfen, ob der Benutzer die notwendigen Berechtigungen hat, könnte sich dabei als problematisch erweisen. Typischerweise werden in Flask dafür Decorators genutzt, was aber nur Sinn macht, wenn dieselbe Logik für möglichst viele Funktionen verwendet werden kann.

## 6. Dashboard-Erweiterung

Im Anschluss an die Migration soll das Benutzerdashboard von Indico noch um für den Benutzer relevante Vorschläge erweitert werden.

### 6.1. Aktueller und gewünschter Zustand

Derzeit zeigt das Dashboard aktuelle Events an, mit denen der User verbunden ist, d.h. an denen er teilnimmt, die er organisiert oder an deren *Paper-Reviewing*-Prozess er beteiligt ist. Neben diesen Events sieht er eine Liste aller Kategorien, in denen er Manager-Rechte hat oder die auf seiner Favoritenliste stehen, und welche Events aktuell in diesen Kategorien stattfinden. Bei dem gesamten Dashboard handelt es sich um ein noch sehr neues Feature, welches aber bereits großen Zuspruch durch die Indico-Benutzer findet.

Zusätzlich zu diesen Informationen wäre es hilfreich, wenn dem Benutzer sowohl potenziell interessante Kategorien als auch weitere Events vorgeschlagen werden könnten. Benutzer sind solche Vorschläge bereits aus größeren Onlineshops gewohnt und gerade in einer großen Indico-Installation mit sehr vielen Events und Kategorien ist die Chance groß, dass solche Vorschläge auch wirklich hilfreich sind und nicht nur auf Events verweisen, die der User bereits kennt.

### 6.2. Kategorien

Alle Events in Indico sind in genau einer Kategorie, die wiederum Teil einer Baumstruktur ist. Dabei werden auf der Indico-Startseite die in der Rootkategorie enthaltenen Kategorien angezeigt und in jeder Kategorie werden entweder die enthaltenen Events oder Unterkategorien aufgelistet. Um Kategorien sinnvoll vorschlagen zu können, muss jeder Kategorie ein Score zugewiesen werden, die die Relevanz dieser Kategorie für den jeweiligen Benutzer angibt. Wenn dieser Score entsprechend hoch ist, wird die Kategorie im Dashboard als Vorschlag angezeigt.

Da Kategorien in Indico sehr unterschiedliche Events enthalten können, macht es wenig Sinn, komplett neue Kategorien vorzuschlagen. Stattdessen bieten sich Kategorien an, mit denen der Benutzer bereits in Kontakt gekommen ist, weil er an einem darin enthaltenen Event teilgenommen hat. Dies alleine ist jedoch nicht ausreichend, um eine Kategorie vorzuschlagen. Gerade bei Benutzern, die Indico schon länger verwenden, ist die Chance groß, dass sie in einigen Kategorien an einem Event teilgenommen haben und seitdem nichts mehr mit dieser Kategorie zu tun hatten. Solch eine Kategorie vorzuschlagen würde also letztendlich nur dazu führen, dass der Benutzer sich über die schlechten Vorschläge ärgert und sie daraufhin grundsätzlich nicht mehr beachtet. Ein gutes Relevanzkriterium ist also, an wie vielen aktuellen Events in der jeweiligen Kategorie der User bereits teilgenommen hat. Da dies jedoch durch längere Pausen ohne Teilnahme negativ beeinflusst wird, selbst wenn im aktuellsten Block 100% Teilnahme herrscht, sollten dabei alle älteren Blöcke verworfen werden, sodass nur das erste Event im aktuellsten Block als frühestes Teilnahmedatum zählt. Um Kategorien auszufiltern, in denen der User regelmäßig an Event teilgenommen hat, dies nun aber nicht mehr tut, kann der Score abhängig von der Anzahl Events, die nach der letzten Teilnahme stattfinden, verringert oder erhöht werden.

Damit zeigen sich bereits ziemlich gute Ergebnisse, allerdings tauchen weiterhin Kategorien auf, die die bisherigen Kriterien erfüllen, aber nicht mehr aktiv genutzt werden und somit keine neuen Events enthalten. Um diese nicht mehr vorzuschlagen, kann der Score abhängig davon, wie weit das neueste Event in der Vergangenheit liegt, verringert werden. Je älter das Event ist, desto weniger relevant ist die Kategorie für den Benutzer.

Unabhängig von der vergangenen Aktivität des Benutzers ist eine Kategorie, an deren Events man bereits lange im Voraus teilnimmt, ein sehr guter Kandidat. Daher erhöht jedes solche Event den Score der Kategorie. Abhängig davon, wie weit das Event in der Zukunft liegt, wird der Score ebenfalls erhöht - ein in Kürze stattfindendes Event ist ein gutes Indiz dafür, dass der User Indico aktiver nutzt und evtl. die Kategorie des Events seinen Favoriten hinzufügen will.

### 6.3. Events

Anders als bei Kategorien ist es bei Events durchaus sinnvoll, „ähnliche“ Events vorzuschlagen. Diese Ähnlichkeit kann auf verschiedene Wege bestimmt werden. Da der

Use-Case, ähnliche „Artikel“ vorzuschlagen, insbesondere in Onlineshops häufig auftritt, existieren bereits einige fertige Lösungen. Es bietet sich also an, zu überprüfen, ob eine solche Lösung für Indico Sinn macht und somit den mit einer Eigenentwicklung verbundenen Aufwand zu vermeiden.

Da das Backend von Indico selbst komplett in Python geschrieben ist, sollte auch die Recommendation Engine entweder auf Python basieren oder entsprechende Python-Bindings besitzen, sodass sie einfach integriert werden kann. Ebenfalls ist es wichtig, dass sie, genau wie Indico, Open Source ist und keine schwierig zu erfüllenden Systemanforderungen hat. Ein Beispiel dafür wäre eine Software, die zwingend eine kommerzielle Oracle-Datenbank benötigt.

### 6.3.1. Crab

*Crab*<sup>28</sup> ist ein *Recommender Framework*, welches auf den Libraries *numpy* und *scipy* basiert. Dabei handelt es sich um Python-Bibliotheken, die diverse mathematische Standardalgorithmen und Hilfsfunktionen zur Verfügung stellen und dank in C geschriebenen Erweiterungen auch in einer Scriptsprache wie Python sehr hohe Performance bieten.

Auf die verwendeten Algorithmen näher einzugehen würde den Umfang dieses Kapitels sprengen. Daher wird nur die eigentliche Funktionalität von Crab betrachtet und inwiefern sie für Indico geeignet ist. Als Datenbasis wird ein Mapping benötigt, welches Benutzer mit Events verknüpft, wobei jede Verknüpfung ein bestimmtes Gewicht hat. Anhand dieser Daten berechnet Crab nun die Ähnlichkeit zwischen den einzelnen Usern. Der Recommender kann schlussendlich mithilfe dieser Daten und den Initialdaten Events vorschlagen, mit denen ein ähnlicher Benutzer eine Verbindung hat.

Die API von Crab ist sehr einfach zu nutzen, allerdings spricht ein wichtiger Faktor gegen eine Verwendung in Indico. Unabhängig von den genutzten Algorithmen benötigt Crab alle Daten gleichzeitig im Speicher. Selbst wenn man ältere Events nicht mit einbeziehen würde, wären dennoch Daten für sehr viele Benutzer zu laden. Mit der CERN-Datenbank wären das ungefähr 50000 Benutzer und deutlich mehr Events. Da Crab keine temporären Dateien anlegt, müssen jegliche Berechnung bei weiteren Durchläufen erneut ausgeführt werden und somit genauso rechen- und speicherintensiv.

---

<sup>28</sup><https://github.com/muricoca/crab/>

### 6.3.2. Probleme

Wenn man einmal näher betrachtet, wie größere Indico-Installationen meist genutzt werden und welche Events relevant sind, bemerkt man schnell, dass sich das Problem stark von dem zuvor erwähnten Use-Case in einem Onlineshop unterscheidet:

- In einem Shop kann ein Benutzer oftmals Artikel bewerten oder entscheidet sich dafür, einen Artikel zu kaufen oder ihn in den Warenkorb zu legen bzw. vor dem Kauf wieder daraus zu entfernen. Mit diesen Informationen lässt sich verhältnismäßig einfach eine Gewichtung zwischen Kunden und Artikeln aufbauen. In Indico ist dies jedoch nicht der Fall - bei Events vom Typ „Lecture“ oder „Meeting“ wird häufig nicht einmal eine Teilnehmerliste geführt, weshalb nur aktiv teilnehmende Benutzer, die beispielsweise durch einen im Zeitplan festgehaltenen Beitrag eine sichtbare Verknüpfung mit dem Event haben.
- Prinzipiell sind zunächst alle Artikel in einem Shop für einen Benutzer relevant bzw könnten es sein. Eine große Indico-Installation enthält jedoch sehr viele unterschiedliche Events. Im Beispiel der CERN-Installation geht das von Human-Resources-Meetings über Meetings der diversen Physik-Experimente bis hin zu Meetings der Systemadministratoren. Obwohl es durchaus möglich sein kann, dass jemand dort an mehreren Meetings teilnimmt, weil er z.B. Systemadministrator bei einem der Experimente ist, führt dies nicht zwangsläufig dazu, dass Events aus beiden Kategorien relevant für seine Kollegen sind.

An diesen Punkten sieht man deutlich, dass eine fertige Recommendation Engine, die mit größter Wahrscheinlichkeit auf den Onlineshop-Use-Case zugeschnitten ist, in Indico nicht wirklich hilfreich ist.

### 6.3.3. Alternative

Deutlich sinnvoller sind Vorschläge zu Konferenzen. Diese finden in der Regel höchstens jährlich statt und dementsprechend nützlich ist ein Hinweis, dass eine bestimmten Konferenz, an der man zuvor bereits einmal teilgenommen hat, wieder stattfindet. Oftmals haben solche jährlich stattfindenden Konferenzen jeweils denselben Titel mit der entsprechenden Jahreszahl am Ende.

Technisch lassen sich auf ähnlichen Titeln basierende Vorschläge sehr einfach realisieren. Indico enthält bereits Plugins, um alle Events einer externen Suchmaschine

zugänglich zu machen. Wenn man nun also automatisiert nach Events mit Titeln sucht, die Ähnlichkeit mit den Titeln bereits besuchter Events bzw. Konferenzen haben und noch nicht stattgefunden haben, ist die Chance groß, dass es sich um genau die gewünschten Vorschläge handelt. Die einzige Schwierigkeit liegt daran, die Titel der bestehenden Events auszulesen ohne alle mit dem Benutzer verbundenen Events einbeziehen zu müssen. Während dies zwar problemlos möglich ist, wenn man Meetings grundsätzlich nicht einbezieht, hat diese Lösung den Nachteil, einen Großteil des Datenbestandes zu ignorieren. Einfacher ist es, die Liste der Events des Users zu filtern und ähnliche Events zusammenzufassen bzw. Duplikate zu entfernen. Insbesondere regelmäßige Meetings haben meist denselben Namen, da sie einfach nur kopiert werden, und sofern Jahreszahlen oder andere Datumsangaben im Titel sind, kann man sie ebenfalls leicht entfernen. Damit hat man eine Liste mit einzigartigen Eventtiteln, die man an die Suchmaschine übergeben kann.

Die Ergebnisse dieser Suche sind jedoch noch nicht sofort für Vorschläge geeignet. Jede Fuzzy-Suche enthält Ergebnisse, die man nicht erwarten würde, da kein Suchalgorithmus unfehlbar ist. Während an sich einzelne unpassende Vorschläge kein Problem sind, dürfen grundsätzlich keine Events angezeigt werden, auf die der Benutzer keinen Zugriff hat - je nach Event könnte bereits der Titel und eine eventuell vorhandene Beschreibung sensible Informationen enthalten. Darüber hinaus hat die Suchmaschine keine Informationen darüber, an welchen Events der Benutzer bereits teilnimmt. Diese müssen daher ebenfalls in Indico ausgefiltert werden.

## 6.4. Aktueller Stand

Der derzeitige *master*-Entwicklungszweig von Indico enthält noch keinerlei Code für Vorschläge, allerdings ist das auf dem in Abschnitt 6.2 beschriebenen Algorithmus basierte Vorschlagssystem für Kategorien fertiggestellt und kann nach weiteren Tests in *master* übernommen werden. Die Implementation ist dabei darauf ausgelegt, auch mit sehr vielen Benutzern performant zu arbeiten. Dies wurde dadurch realisiert, dass nur Benutzer, die das Dashboard auch nutzen, Vorschläge erhalten. Darüber hinaus werden die Vorschläge nachts in einem separaten Prozess generiert, sodass sie nicht während der Hauptarbeitszeit tagsüber die Datenbank belasten.

Für Event-Vorschläge existiert ein einfacher Prototyp, der aktuelle Events in den favorisierten Kategorien des Benutzers durchsucht und Events vorschlägt, deren Teilnehmerlisten zu mindestens 25% übereinstimmen oder denselben Vorsitzenden

haben. Jedoch haben sich die damit gefundenen Vorschläge als wenig hilfreich herausgestellt, da sie hauptsächlich regelmäßig stattfindende Meetings in der Ergebnisliste hatten, die aber bereits aufgrund der favorisierten Kategorie im bestehenden Dashboard angezeigt werden.

Manuelle Tests mit der am CERN genutzten Indico-Suchmaschine lassen jedoch darauf schließen, dass titelbasierte Vorschläge nützlich sind und der Entwicklungsaufwand sich daher lohnt.



## 7. Fazit

Im Rahmen dieser Masterarbeit am CERN wurden nicht nur die beiden zunehmend verbreiteteren Python-Webframeworks *Django* und *Flask* unter die Lupe genommen, sondern auch der frameworkähnliche Code von Indico untersucht. Bei allen Frameworks haben sich sowohl Stärken als auch Schwächen gezeigt, wobei klar erkennbar wurde, dass der Nutzen vieler Features stark von der jeweiligen Anwendung abhängt. So ist das ORM-System von Django für eine Anwendung wie Indico nicht von Nutzen, sofern man sie nicht von Grund auf neu entwickelt, aber wenn man eine weniger umfangreiche Webapplikation von Anfang an auf Django basiert, ist ein integriertes Datenbankframework durchaus nützlich.

Alle mit Flask verbundenen Änderungen einschließlich der sauberen URLs sind inzwischen im Hauptentwicklungszweig von Indico integriert und dokumentiert, sodass andere Indico-Entwickler von den Neuerungen profitieren können. Im Laufe der Entwicklung hat sich gezeigt, dass es sehr viele Bereiche in Indico gibt, die durch Flask-Features verbessert werden könnten, aber um den Übergang auch für andere Entwickler möglichst einfach zu gestalten, noch nicht umgesetzt wurden. Dies ist leicht zu erklären, indem man das Entwicklungsmodell bei der Nutzung von Git betrachtet. Jedes Feature wird in einer separaten Branch des Codes entwickelt. Dies kann prinzipiell jederzeit aktualisiert werden, sodass alle eigenen Änderungen auf der aktuellsten *master*-Version basieren. Bei größeren Änderungen wie einer Frameworkmigration sind Konflikte bei diesen Updates jedoch ausgesprochen wahrscheinlich und müssen jeweils manuell behoben werden. Daher war es erwünscht, die Migration möglichst schnell soweit abzuschließen, sodass der Code in den *master*-Branch integriert werden konnte.

Ebenfalls hat sich gezeigt, wie hilfreich eine vollständige Testabdeckung ist. Eine große Anwendung wie Indico manuell zu testen ist sehr aufwändig.

Langfristig wäre es aus Entwicklersicht natürlich angenehm, wenn alle Bestandteile von Indico Flask-Features nutzen würden und jeglicher alte Code entfernt würde. Dies ist jedoch aufgrund des enormen Aufwands, der damit verbunden ist, nicht zu erwarten.

Die Entwicklung der *Recommendation Engine* für Kategorien und die Planung einer solchen für Events hat gezeigt, dass ein gerade aus Onlineshops eigentlich als

alltglich erscheinendes Feature alles andere als einfach zu implementieren ist und die Tatsache, dass Indico kein Onlineshop sondern ein *Conference Management System* ist, die Aufgabe nicht einfacher macht. Da das existierende Dashboard jedoch sehr positiv aufgenommen wurde ist damit zu rechnen, dass auch ein relativ simples Vorschlagssystem fr Kategorien gut aufgenommen wird.

# Literaturverzeichnis

- [Eby03] Phillip J. Eby. PEP 333 - python web server gateway interface v1.0. <http://www.python.org/dev/peps/pep-0333/>, December 2003.
- [Fas03] Karl Fast. Python mailing list: What is 'pythonic'? <http://mail.python.org/pipermail/tutor/2003-October/025932.html>, 2003.
- [Fer08] Pedro Ferreira. Indico developer's guide for the newbie. <http://indico.cern.ch/materialDisplay.py?materialId=slides&confId=37937>, 2008.
- [HV06] Peter A. Henning and Holger Vogelsang. *Handbuch Programmiersprachen: Softwareentwicklung zum Lernen und Nachschlagen*. Hanser Verlag, 2006.
- [Ind13] Indico. Indico project homepage. <http://indico-software.org/>, 2013. [12. August 2013].
- [Pet04] Tim Peters. PEP 20 - the zen of python. <http://www.python.org/dev/peps/pep-0020/>, August 2004.
- [RC04] D. Robinson and K. Coar. The Common Gateway Interface (CGI) Version 1.1. RFC 3875 (Informational) <http://www.ietf.org/rfc/rfc3875.txt>, October 2004.
- [RC12] Armin Ronacher and Nick Coghlan. PEP 414 - explicit unicode literal for python 3.3. <http://www.python.org/dev/peps/pep-0414/>, February 2012.
- [vR09] Guido van Rossum. The history of python: First-class everything. <http://python-history.blogspot.de/2009/02/first-class-everything.html>, 2009.
- [Whe07] David A. Wheeler. The Free-Libre / Open Source Software (FLOSS) license slide. <http://www.dwheeler.com/essays/floss-license-slide.html>, September 2007. [28. August 2013].
- [Wik13] Python Wiki. Should i use python 2 or python 3 for my development activity? <http://wiki.python.org/moin/Python2orPython3>, 2013.

# A. Anhang

## A.1. Screenshots von Indico

**INDICO**  
Integrated Digital Conference

Home Create event Room booking Administration My profile Help

### Main categories

Welcome to Indico. The Indico tool allows you to manage complex conferences, workshops and meetings. In order to start browsing, please select one of the categories below.

CERN-Related Clubs, Associations and Forums	1,973 events
Committees	2,671 events
Conferences, Workshops and Events	3,239 events
Departments	19,052 events
Experiments	194,407 events
Projects	15,369 events
Schools, Seminars and Courses	8,832 events
TEST Category	2,913 events

**News**

**Interface improvements - Mobile, Dashboard an...**  
Posted on 23/04/2013  
Check out the latest features!  
Posted on 09/03/2013

**Upcoming events**

**Radiation from Relativistic Electrons in Periodic...**  
ongoing till Friday 18:50

**TWEPP 2013 - Topical Workshop on Electronic...**  
ongoing till Friday 18:00

**16eme Forum Utilisateurs CATIA au CERN**  
starts Thursday 9:00

**Implications of the Higgs discovery for Dark Mat...**  
starts Thursday 14:00

**Muon Accelerators for the Next Generation of H...**  
starts Thursday 14:15

**Non-Local Bias factors: Theory vs simulations**  
starts Friday 11:30

Powered by Indico

Abbildung A.1: Startseite von Indico

Your events at hand	Your categories	
17 Sep 2013 Indico developers review meeting	CIS Departments >> IT >> Groups	★
18 Sep 2013 AVC section meeting	IT Departments	★
Today Indico developers review meeting	Indico Departments >> IT >> Groups >> CIS >> Audio Visual & Conferencing Services	★
Tomorrow AVC section meeting	Section Meetings Departments >> IT >> Groups >> CIS >> Audio Visual & Conferencing Services	★
14 Oct 2013 20th International Conference on Compu...		
13 Dec 2013 Meeting JB		
30 May 2014 Test Vidyo and Reg files 1		

Happening in your categories
Now Third General Assembly Helix Nebula Initiative Helix Nebula
Today ATLAS-IT Hadoop meeting CASTOR
Today IT-SDC Section Leaders Meeting Section Leaders Meetings

Abbildung A.2: Benutzerdashboard (ohne Vorschläge)

## A.2. Quellcodes

### A.2.1. AST einer mod\_python-Funktion

```

1 FunctionDef(
2     name='index',
3     args=arguments(args=[Name(id='req', ctx=Param())],
4         vararg=None, kwarg='params', defaults=[]),
5     body=[
6         Return(
7             value=Call(
8                 func=Attribute(
9                     value=Call(
10                        func=Attribute(
11                           value=Name(id='rhAbout',
12                               ctx=Load()),
13                           attr='RHAbout', ctx=Load())
14                        ),
15                        args=[Name(id='req', ctx=Load())],
16                        keywords=[],
17                        starargs=None,
18                        kwargs=None),
19                        attr='process',
20                        ctx=Load()),
21                        args=[Name(id='params', ctx=Load())],
22                        keywords=[],
23                        starargs=None,
24                        kwargs=None
25                    )
26                )
27    ],
28    decorator_list=[]
29 )

```

**Listing A.1:** about.py - Abstract Syntax Tree

### A.2.2. Algorithmus für Kategorie-Scores

---

```
1 def _get_category_score(avatar, categ, attended_events):
2     idx = IndexesHolder().getById('categoryDateAll')
3     attended_events_set = set(attended_events)
4     # We care about events in the whole timespan where the
        user attended some events. However, this might
        result in some missed events e.g. if the user was
        not working for a year and then returned. So we
        throw away old blocks (or rather adjust the start
        time to the start time of the newest block)
5     first_event_date =
        attended_events[0].getStartDate().replace(hour=0,
        minute=0)
6     last_event_date =
        attended_events[-1].getStartDate().replace(hour=0,
        minute=0) + timedelta(days=1)
7     blocks =
        _get_blocks(_unique_events(idx.iterateObjectsIn(
8         categ.getId(), first_event_date,
        last_event_date
9         )), attended_events_set)
10    for a, b in _window(blocks):
11        # More than 3 months between blocks? Ignore the
            old block!
12        if b[0].getStartDate() - a[-1].getStartDate() >
            timedelta(weeks=12):
13            first_event_date =
                b[0].getStartDate().replace(hour=0,
                minute=0)
14
15        # Favorite categories get a higher base score
16        favorite = categ in avatar.getLinkTo('category',
            'favorite')
17        score = 1 if favorite else 0
18        # Attendance percentage goes to the score directly. If
            the attendance is high chances are good that the
```

```
        user is either very interested in whatever goes on
        in the category or it's something he has to attend
        regularly.
19     total = sum(1 for _ in
        _unique_events(idx.iterateObjectsIn(categ.getId(),
        first_event_date, last_event_date)))
20     attended_block_event_count = sum(1 for e in
        attended_events_set if e.getStartDate() >=
        first_event_date)
21     score += attended_block_event_count / total
22     # If there are lots/few unattended events after the
        last attended one we also update the score with that
23     total_after = sum(1 for _ in
        _unique_events(idx.iterateObjectsIn(categ.getId(),
        last_event_date + timedelta(days=1), None)))
24     if total_after < total * 0.05:
25         score += 0.25
26     elif total_after > total * 0.25:
27         score -= 0.5
28     # Lower the score based on how long ago the last
        attended event was if there are no future events.
        We start applying this modifier only if the event
        has been more than 40 days in the past to avoid it
        from happening in case of monthly events that are
        not created early enough.
29     days_since_last_event = (date.today() -
        last_event_date.date()).days
30     if days_since_last_event > 40:
31         score -= 0.025 * days_since_last_event
32     # For events in the future however we raise the score
33     now_local = utc2server(nowutc(), False)
34     attending_future = [e for e in
        _unique_events(idx.iterateObjectsIn(categ.getId(),
        now_local, last_event_date)) if e in
        attended_events_set]
35     if attending_future:
36         score += 0.25 * len(attending_future)
```



```
37     days_to_future_event =  
        (attending_future[0].getStartDate().date() -  
         date.today()).days  
38     score += max(0.1, -(max(0, days_to_future_event -  
        2) / 4) ** (1 / 3) + 2.5)  
39     return score
```

---

**Listing A.2:** Der für Kategorievorschläge genutzte Scoringalgorithmus