Projektarbeit an der Hochschule-Bochum

Technische Dokumentation KOMPLETT

Kooperation mittels einer Plattform für Einzelspieler & Tennisvereine und kommerzielle Tennishallen

Wroblewski, Gürkan, Sahm, Derksen
14.01.2015

Inhaltsverzeichnis

1. Einleitung

KOMPLETT ist eine Webapplikation die in Python, HTML/CSS und JavaScript programmiert ist. Python – mithilfe des Webframeworks Django – ist die Server-seitige Sprache, zum Client werden durch HTML definierte und mit CSS formatierte Inhalte gesendet. Diese wiederum werden mit der Sprache JavaScript Browser-seitig um dynamische Funktionalitäten erweitert.

Als Webserver dient nginx, welcher auf einem Ubuntu-VPS läuft. Zur Kommunikation zwischen nginx und Django wird die Middleware gunicorn verwendet. Gleichzeitig läuft auf dem Ubuntu-Server eine PostgreSQL-Datenbank, welche direkt von Django angesprochen wird.

2. Ubuntu-VPS

Ausgeführt und dem Web zur Verfügung gestellt, wird die Applikation über einen Ubuntu-Virtual-Private-Server. Dies ist eine angemietete KVM-Instanz eines Ubuntu-Hosts. KVM ist eine Linux Virtualisierungstechnik und wird verwendet, um über einen mit Linux bestückten Server mehrere Betriebssysteminstanzen zur Verfügung zu stellen. Der Vorteil für den Betreiber ist natürlich, dass er viele Instanzen über einen Server anbieten kann. Der Kunde kann nun günstig eine virtuelle Maschine anmieten und die zugeteilten Rechenressourcen, wie Anzahl der CPU-Kerne, RAM und Festplattenplatz nach Bedarf dynamisch skalieren. KOMPLETT läuft auf einem virtualisierten Ubuntu bei dem Anbieter DigitalOcean, welcher sich auf Linux-Virtualisierung spezialisiert hat und VPS-Instanzen mit SSD-Festplatte zu günstigen Preisen anbietet. Es gibt die Möglichkeit einen europäischen Server-Standort zu wählen.
Die Administration des Servers erfolgt wahlweise über VNC (das Remote-Desktop-Protokoll von Linux-Maschinen) oder ssh. Letzteres steht für secure shell und bietet einen sicheren, verschlüsselten Zugriff auf die Kommandozeile.

Nach Einrichtung des Servers wird das automatisch erstellte Passwort für den Benutzer root an die im Benutzeraccount hinterlegte E-Mail-Adresse gesendet. Dann ist die Anmeldung am Server möglich:

```
ssh root@178.62.195.241
```

Sofort nach dem ersten Login muss das root-Passwort geändert werden:

```
root@178.62.195.241's password:
You are required to change your password immediately (root enforced)
Welcome to Ubuntu 14.10 (GNU/Linux 3.16.0-23-generic x86_64)

Changing password for root.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
root@KOMPLETT:~#
```

Hierbei wird das Passwort nicht angezeigt. Danach werden fail2ban und die ufw Firewall (siehe 3.1) konfiguriert sowie der Server neugestartet.
Weiter müssen alle Pakete aktualisiert sowie der Webserver nginx, gunicorn und die benötigten Python-Programmbibliotheken inklusive Django installiert werden.

3. Absicherung des Servers

Um den Server abzusichern, werden folgende Maßnahmen ergriffen:

1. Absicherung durch ufw – uncomplicated Firewall
2. Verhindern von root-Login über ssh
3. Fail2ban
4. Verwenden sicherer Passwörter
5. Beständige Updates aller Pakete
6. Überprüfen der Logdatei auth.log

Nun zu den einzelnen Punkten:

3.1 Absicherung durch Firewall

Der Server wird durch ufw – das ist die uncomplicated firewall – abgesichert. UFW ist ein Kommandozeilen-Frontend für das Linux Kernel Modul iptables. Anstatt einzelne iptables Regeln einzugeben und diese zu persistieren, kann mit ufw eine sehr viel komfortablere Konfiguration vorgenommen werden. Mit

```
root@KOMPLETT:~# sudo ufw default deny incoming
Default incoming policy changed to 'deny'
```

wird das Standardverhalten von iptables für ankommende Pakete auf deny gesetzt. Für unseren Webserver müssen wir die Firewall für zwei Dienste öffnen: ssh (Port 22), um den Server zu administrieren und http (Port 80) sowie https (Port 443), um die KOMPLETT-Website erreichbar zu machen:

```
root@KOMPLETT:~# sudo ufw allow ssh, http , https
```

Jetzt erlauben wir noch ausgehende Verbindungen und die Firewall ist bereits fertig konfiguriert.

3.2 Verhindern von root-Login über ssh

Den root-Login über ssh zu verhindern bedeutet für einen Angreifer, dass er neben dem Passwort auch einen ihm unbekannten Benutzernamen erraten muss. Dafür muss ein alternativer Useraccount erstellt werden, welcher dann für ssh Logins zuständig ist:

```
root@KOMPLETT:/etc# adduser flambierkuchenx3
Adding user `flambierkuchenx3' ...
```

Dem hinzugefügten User müssen nun root-Rechte zuerkannt werden, dazu wird die Datei /etc/sudoers.tmp geöffnet und unter der Zeile

```
# User privilege specification
root    ALL=(ALL:ALL) ALL
```

wird folgende Zeile hinzugefügt:

```
flambierkuchenx3 ALL=(ALL:ALL) ALL
```

Der User flambierkuchenx3 hat jetzt alle Rechte des root-Accounts. Um dem eigentlichen root-User den Login über ssh zu verbieten, öffnen wir mit

```
root@KOMPLETT:/etc# nano /etc/ssh/sshd_config
```

die SSH-Konfigurationsdatei. Wir setzen

```
PermitRootLogin no
```

und erlauben explizit nur dem erstellen User das Zugriffsrecht:

```
AllowUsers flambierkuchenx3.
```

Nach dem Neustart des ssh-Dienstes sind die Regeln aktiv und ein Login-Versuch mit root schlägt fehl.

3.3 fail2ban

Fail2ban ist ein Werkzeug, welches den Einbruch in den Server über ssh erschwert. Fail2ban kann so konfiguriert werden, dass nach einer definierten Anzahl an fehlgeschlagenen Login-Versuchen über ssh automatisch die Firewall-Konfiguration so angepasst wird, dass die betreffende IP-Adresse für eine definierte Zeit gesperrt wird.
Fail2ban muss installiert werden:

```
root@KOMPLETT:~# sudo apt-get install fail2ban
```

In der Konfigurationsdatei kann der Dienst konfiguriert werden. Die wichtigsten Einstellungen sind bantime (=Zeit die ein Angreifer gesperrt wird) und maxretry (=Maximale Einlogmöglichkeiten innerhalb eines definierten Zeitrahmens). Der KOMPLETT-Server wurde folgendermaßen konfiguriert:

```
# "bantime" is the number of seconds that a host is banned.
bantime  = 60000

# A host is banned if it has generated "maxretry" during the last
"findtime"
# seconds.
findtime = 600
maxretry = 3
```

Fail2ban ist eine essentiell wichtige Absicherung des ssh-Dienstes.

3.4 Verwenden sicherer Passwörter

Für die Accounts auf dem Server werden sichere Passwörter mit einer Länge von mindestens 20 Zeichen gewählt. Die Passwörter sind automatisch und zufällig generiert und beinhalten Buchstaben, Zahlen und Sonderzeichen.

3.5 Beständige Updates aller Pakete

Der Server-Administrator muss durch regelmäßige Updates der installierten Pakete sicherstellen, dass keine Sicherheitslücken alter Software-Versionen ausgenutzt werden können.

3.6 Beständige Überprüfung der Logdatei auth.log

In der Logdatei auth.log unter /var/log/ werden alle Anmeldeversuche am System protokolliert. Diese sollte der Administrator mit einer lokalen Dokumentation seiner Login-Versuche abgleichen und bei Abweichungen geeignete forensische Maßnahmen ergreifen.

4. Python

Die Webapplikation ist serverseitig in Python programmiert. Python wird meist – ähnlich wie Java – in Bytecode übersetzt und in einer virtuellen Laufzeitmaschine ausgeführt. Python ist im Gegensatz zu Java nicht typisiert, eine Variable muss nicht mit einem Datentyp deklariert werden und Datentypen können implizit ineinander umgewandelt werden. Zu einer gültigen Python-Syntax gehört auch die korrekte Verwendung von Whitespaces bzw. Einrückungen. Dies ist gültige Python-Syntax:

```
1.  # Class to call and create opening_times
2.  class OpeningTimeFilterList(generics.RetrieveUpdateDestroyAPIView):
3.      queryset = OpeningTime.objects.all()
4.      serializer_class = OpeningTimeSerializer
5.      filter_fields = ('opening_time_id', 'title', 'start', 'end', 'user')
```

Derselbe Code mit einer falschen Einrückung bei queryset (Zeile 3) ist ungültig:

```
1.  # Class to call and create opening_times
2.  class OpeningTimeFilterList(generics.RetrieveUpdateDestroyAPIView):
3.  queryset = OpeningTime.objects.all()
4.      serializer_class = OpeningTimeSerializer
5.      filter_fields = ('opening_time_id', 'title', 'start', 'end', 'user')
```

Python ist standardmäßig in den meisten Linux-Distributionen installiert und kann somit sofort verwendet werden.

5. Virtualenv

Virtualenv ist ein Python-Modul, welches zur Einrichtung voneinander isolierten Python-Laufzeitumgebungen dient. Zusätzliche Programmbibliotheken werden jetzt nur in dieser lokalen Umgebung installiert und nicht betriebssystemweit. Somit können mehrere Projekte auf dem gleichen Server realisiert werden, ohne dass sich Bibliotheksabhängigkeiten in die Quere kommen. Virtualenv ist nicht standardmäßig mit Python installiert, muss also noch eingebunden werden.

6. Django

Um Low-Level Aufgaben, wie die Verarbeitung von HTTP-Requests, nicht von Grund auf programmieren zu müssen, verlässt man sich in der Webentwicklung auf Frameworks. Ein Framework ist eine Klassenbibliothek mit Klassen und deren Methoden für immer wieder aufkehrende Aufgaben bei der Web-Entwicklung. Wir haben uns für das Django-Framework entschieden, weil es sehr umfangreich und sehr gut dokumentiert ist. Das Framework bietet unter anderem die Möglichkeit, das ER-Modell direkt in Klassen auszuprogrammieren. Django sorgt dann automatisch für die Übersetzung in SQL-Befehle. Man kann also direkt auf Objekten und deren Attributen arbeiten und muss keine SQL-Befehle schreiben. Dies nennt man Object-Relational-Mapping, also das Umsetzen von Programmobjekten auf eine relationale Datenbank und umgekehrt. Django hält sich lose an das Model-View-Controller Pattern, neben der Möglichkeit, sich um die Models, also die SQL-Tabellen zu kümmern, steht also ein Controller für das URL-Routing zur Verfügung sowie die Möglichkeit, dynamischen Inhalt durch Views zu generieren.
Django wird in der erstellten virtuellen Umgebung installiert, dazu muss zuerst die virtualenv aktiviert werden:

```
deploy@KOMPLETT:/$ source /komplett/bin/activate
(komplett) deploy@KOMPLETT:/$
```

Nun befinden wir uns in der virtuellen Umgebung und können Pakete installieren:

```
(komplett) deploy@KOMPLETT:/$ pip install Django
```

Nach Einrichtung muss das Django-Projekt über git vom Repository geladen werden.

7. Nginx und gunicorn

Auf dem Webserver-Port der virtuellen Maschine hört der leichtgewichtige und leicht konfigurierbare Webserver nginx. Nginx läuft inzwischen auf über 20% der weltweiten Webserver und ist somit nach Apache und Windows IIS die drittgrößte Kraft auf dem Webserver Markt. Nginx empfängt nun die HTTP-Anfragen und leitet diese an die Django-Applikation weiter. Um das zu bewältigen, fungiert gunicorn als Middleware zwischen nginx und Django. Gunicorn ist ein in Python geschriebener WSGI-konformer HTTP-Server, welcher für die Kommunikation zwischen Django und nginx zuständig ist. WSGI ist eine Interface-Spezifikation, welche die Kommunikation zwischen Applikation (Django) und Server (nginx) beschreibt. Nginx wird auf Betriebssystemebene installiert, während gunicorn nur in der virtuellen Umgebung läuft.

8. Datenbank

Als Datenbank wird PostgreSQL verwendet. PostgreSQL ist robust und erprobt und die bevorzugte Datenbank im Zusammenspiel mit Django. Mit

```
root@KOMPLETT:~# sudo apt-get install postgresql
```

wird PostgreSQL installiert. Danach muss ein Nutzer sowie eine Datenbank angelegt werden und der Nutzer muss vollen Zugriff auf diese Datenbank bekommen:

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE komplett TO komplett
```

Danach ist die Datenbank bereit, von Django genutzt zu werden.

9. HTML/CSS

HTML ist eine Auszeichnungssprache und wird benutzt, um die Grundstruktur und den Inhalt einer Website zu bestimmten. Mit CSS wiederum wird diesen Inhalten ein Design verliehen, die Cascading Style Sheets sind somit für das Aussehen verantwortlich. So kann eine saubere Trennung von Inhalt und Design stattfinden. Django bietet eine Template-Sprache mit einfachen Anweisungen, wie Bedingungen und Schleifen. Im HTML-Dokument wird so auf Objekte zugegriffen, die von Django (also vom Applikationsserver) an das HTML-Dokument geschickt worden sind, um dieses mit dynamischem Inhalt zu füllen. Nachfolgend sei ein Beispiel der Zusammenarbeit zwischen Model, View, Controller und Template-Sprache im HTML:

Model:

```
1.  # Tennisplatz
2.  class TennisCourt(models.Model):
3.      court_id = models.AutoField(primary_key=True)
4.      tennis_club = models.ForeignKey(TennisClub)
5.      local = models.ForeignKey(Local)
6.      floor_covering = models.ForeignKey(FloorCovering)
7.      name = models.CharField(max_length=50)
8.      slug = models.SlugField()
9.
10.     class Meta:
11.         unique_together = ('tennis_club', 'name')
```

Controller:

```
1.  urlpatterns = patterns('',
2.      ...
3.      url(r'kontrollzentrum/court_list/$', views.court_list, name='court_list'),
4.      ...
5.  )
```

View:

```
@login_required
def court_list(request):
    args = RequestContext(request)
    courtlist=TennisClub.objects.get(pk=request.user.login_id).tenniscourt_set.all()

    args['courts'] = courtlist
    return render_to_response('homepage/tenniscourt_list.html', args)
```

Auszug aus HTML-Dokument:

```
 <div class="container">
    <div class="form-group col-lg-6">
       {% if courts %}
          <ul>
             {% for tenniscourt_set in courts %}
                <li>
                   <a href="{% url 'homepage:court_change' tenniscourt_set.court_id
%}">{{ tenniscourt_set.name }}                    -
                     {{ tenniscourt_set.floor_covering }} </a>
                </li>
             {% endfor %}
          </ul>
       {% else %}
          <strong>Sie haben noch keine Plätze angelegt.</strong>
       {% endif %}
   </div>
</div>
```

Beim Aufruf der URL http://www.komplett.de/kontrollzentrum/court_list/ erkennt der Controller, dass diese URL auf die View – d. h. die Funktion – court_list verweist. Dann wird diese Funktion ausgeführt. In der Funktion wird das Array courtlist mit allen Tenniscourt Instanzen des eingeloggten Users gefüllt. Dann wird dieses Array dem Argument args mit dem Namen courts zugewiesen, dieses Argument wird dann zusammen mit der anzuzeigenden HTML-Datei der Funktion render_to_response übergeben. In der HTML-Datei kann man jetzt mit der Template-Sprache auf das courts Argument zugreifen, in einer Schleife über alle Tenniscourts iterieren und das Ergebnis in einer Liste ausgeben. Das dann fertige HTML-Dokument mit der Liste der Tenniscourts, wird dann beim Benutzer vom Browser angezeigt.

10. Bootstrap

Bootstrap ist ein frei verfügbares CSS-Framework von Twitter. Bootstrap stellt vordefinierte CSS-Elemente zur Verfügung. Zuerst musst Bootstrap in das HTML-Template eingebunden werden:

```
<link rel="stylesheet" href="{% static 'homepage/css/bootstrap.css' %}"/>
```

Eine Seite wird in ein Gitter (Grid) eingeteilt, die größtmögliche Breite ist dabei 12, hier ein Beispiel:

```
<div class="row">
    <div class="col-sm-4"> </div>
    <div class="col-sm-4"> </div>
    <div class="col-sm-4"> </div>
</div>
```

Hier wird eine Zeile mit drei Spalten der Breite 4 definiert. Die Aufteilung könnte auch anders aussehen, zum Beispiel könnte die erste Spalte die Breite 5 und die letzte die Breite 3 haben, nur darf die Gesamtbreite 12 nicht übersteigen.
Aufgrund dieser Gitterstruktur lassen sich Designs von Homepages einfach aufbauen. Weiter bietet Bootstrap viele nützliche Elemente wie Buttons, Navigationselemente etc. Bootstrap wird heute in vielen Webprojekten verwendet und stellt eines der nützlichsten Frameworks für Webdesigner dar.

11. JavaScript

JavaScript ist eine ursprünglich von Netscape entwickelte Sprache, die im Browser des Clients ausgeführt wird. JavaScript hat nichts mit Java zu tun, die Namensgebung war eher ein Marketinggag. Mit JavaScript ist es möglich, der Website dynamische Funktionalitäten zu verleihen. Die Website kann so zum Beispiel auf Benutzereingaben reagieren und das Aussehen dynamisch im Browser verändern, Daten an einen Server senden oder Daten von einer REST-Schnittstelle abrufen.

12. jQuery

JQuery ist eine JavaScript-Klassenbibliothek, welche die DOM-Manipulation und –Interaktion der HTML-Seite vereinfacht. Weiter bietet jQuery viele nützliche und oft gebrauchte Funktionen wie AJAX-Calls (asynchrone Aufrufe an Webservices) und HTML-Elementselektion.

13. REST

REST steht für Representational State Transfer und ist ein Architekturvorschlag für die Kommunikation zwischen Webservices und Clients. REST reduziert Webservices auf fest definierte Ressourcen, zum Beispiel http://www.komplett.de/allespieler und darauf definierten Operation wie GET, POST und PUT. Mit diesen Operationen kann nun auf die Ressource zugegriffen werden, um Daten abzurufen, zu verändern und anzulegen. So kann man Ressourcen und die Verarbeitung dieser Ressourcen in Anwendungen kapseln. Es ist nun möglich, die Datenbankimplementation unter der Ressource zu verändern ohne dass ein Programm, welches auf die Ressource zugreift, angepasst werden muss, weil sich die Operationen nach außen hin nicht verändern.
In der KOMPLETT-Applikation wird ein Kalender-Plugin verwendet, welches eine REST-Schnittstelle erfordert um Events zu speichern und aufzurufen. Dafür wurde das Django-Plugin Djangorestframework benutzt. Dieses bietet die Möglichkeit, Ressourcen (also Models die letztendlich zu SQL-Tabellen werden) über eine URL anzubieten und darauf mit den genannten Operationen zuzugreifen. Nachfolgend ein Beispiel dazu:

Model:

```
1.  # Saisondaten
2.  class WinterSeason(models.Model):
3.
4.      SEASONTYPE = (
5.          ('winter', 'winter'),
6.      )
7.
8.      season_id = models.AutoField(primary_key=True)
9.      user = models.ForeignKey(LoginUser, unique=True)
10.     type = models.CharField(max_length=6, choices=SEASONTYPE)
11.     beginning = models.DateTimeField()
12.     ending = models.DateTimeField()
```

Serializer:

```
1.  class WinterSeasonSerializer(serializers.ModelSerializer):
2.
3.      class Meta:
4.          model = WinterSeason
5.          fields = ('season_id', 'user', 'type', 'beginning', 'ending')
```

View:

```
1.  # Klasse um seasons abzurufen und anzulegen
2.  class WinterSeasonList(generics.ListCreateAPIView):
3.      queryset = WinterSeason.objects.all()
4.      serializer_class = WinterSeasonSerializer
5.      filter_fields = ('type', 'user')
```

Controller:

```
1.  urlpatterns = patterns('',
2.      ...
3.      #Season-Ressourcen ueber Djangorestframework
4.      url(r'^winterseasons/$', views.WinterSeasonList.as_view()),
5.      ...
6.  )
```

Es geht um das Model WinterSeason, welches natürlich eine Entität ist, die in der SQL-Datenbank abgespeichert wird. Über die von dem Djangorestframework bereitgestellte Klasse ModelSerializer wird das Model WinterSeason serialisiert, wobei es möglich ist die Spalten anzugeben, die bereitgestellt werden sollen. Dazu ist eine View-Klasse notwendig, diese Klasse enthält ein Array mit allen WinterSeason-Instanzen und wendet auf diese den darüber definierten Serialisierer an. Gleichzeitig wird definiert, nach welchen Spalten Anfragen gefiltert werden können. Im Controller wird nun nur noch die URL bereitgestellt und mit der View-Klasse verknüpft.
Jetzt ist es möglich, auf der Ressource http://www.komplett.de/winterseasons/ REST-konforme Operationen durchzuführen. Eine GET-Anfrage ohne Filterung liefert alle WinterSeason-Instanzen die in der Datenbank gespeichert sind:

```
0:
{
        season_id: 1
        user: 2
        type: "winter"
        beginning: "2014-10-29T00:00:00"
        ending: "2015-04-15T00:00:00"
},
1:
{
        season_id: 2
        user: 3
        type: "winter"
        beginning: "2014-10-07T00:00:00"
        ending: "2014-10-30T00:00:00"
}
```

usw. In diesem Fall werden die Objekte als JSON-Strings geliefert, weil das in der Anfrage definiert ist. Hier sind auch andere Formate wie XML denkbar. Des Weiteren ist es natürlich möglich auch PUT (Objekt updaten) und POST (Objekt anlegen) Operationen durchzuführen.

14. Fullcalendar Plugin

Um für die Hallenbelegungen einen Kalender darzustellen, wird das auf JavaScript basierende Fullcalendar Plugin verwendet. Dieses Plugin bietet die Möglichkeit, einen Kalender mit verschiedenen Parametern anzuzeigen und Event-Quellen zu definieren. Die Implementation des Backends – also der Speicherung und Verarbeitung dieser Quellen – obliegt dann dem Programmierer. Bei Komplett wurde das oben beschriebene Djangorestframework benutzt, um eine REST-konforme Schnittstelle für den Kalender zur Verfügung zu stellen. Um den Kalender anzuzeigen muss zuerst die Klassenbibliothek in das HTML-Dokument eingebunden werden:

```
<script src="{% static 'homepage/js/fullcalendar.js' %}"></script>
```

Weiter muss an der gewünschten Stelle ein div-Container erstellt werden:

```
<div id="calendar"></div>
```

Über einen jQuery-Selektor kann jetzt, in einer eigens definierten Javascript-Datei, auf das calendar-Element zugegriffen werden:

```
1.  $('#calendar').fullCalendar({
2.
3.                      header: {
4.                      left: 'prev,next today',
5.                      center: 'title',
6.                      right: 'month,agendaWeek,agendaDay'
7.
8.                  },
9.                  droppable: true,
10.                 editable: true,
11.                 deletable: true,
12.                 lang: 'de',
13.                 weekNumbers: true,
14.                 slotDuration: "00:15:00",
```

```
15.
16.                 eventSources: [
17.                     {
18.                         url:'/events/',
19.                         data: { 'user': user_id, 'court': court_id }
20.                     }
21.                 ]
22. });
```

Über den calendar-Selektor wird die fullCalendar-Funktion aus der fullcalendar.js Bibliothek aufgerufen und der Kalender wird im HTML mit den festgelegten Einstellungen angezeigt.

Die Event-Logik wird auf JavaScript-Ebene implementiert. Djangorestframework dient als Backend zur persistenten Speicherung der Eventdaten. Die größte Herausforderung sind die wiederkehrenden Events. Hier wird mittels der mathematischen Funktion ein unique_identifier erzeugt, welcher bei allen Events, die als wiederkehrend zusammenhängen, als gemeinsames Merkmal verwendet wird:

```
unique_identifier = Math.floor((Math.random() * 4000000) + 1);
```

Es wird überprüft, ob der identifier wirklich einmalig ist und dann mit jedem wiederkehrenden Event, welches zur Eventkette gehört gespeichert. Es wird von dem Datum und der Uhrzeit ausgegangen, an dem der User das erste Event einträgt. Darauf basierend werden immer genau sieben Tage auf diesen Zeitstempel addiert und das Event wiederum gespeichert:

```
1. startmoment_moment.add(7,'days');
2. endmoment_moment.add(7,'days');

1. saveEvent(copiedEventObject.title, startmoment, endmoment, copiedEventObject.color,
   user_id, court_id, unique_identifier, season);
```

Die Methode saveEvent speichert die Daten über einen asynchronen Aufruf an die REST-Schnittstelle unseres Servers:

```
1. function saveEvent(title, startmoment, endmoment, color, user_id, court_id, unique_i
   dentifier, season) {
2.          // serialize
3.          JSON.stringify(title, startmoment, endmoment, color, user_id, court_id, uni
   que_identifier, season);
4.          $.ajax({
5.              type: "POST",
6.              url: "/events/",
7.
8.              dataType: "json",
9.              content: "application/json",
10.             data: { 'title': title, 'start': startmoment, 'end': endmoment, 'color'
   : color, 'user': user_id, 'court': court_id,    'unique_identifier': unique_identifi
   er, 'season': season},
11.             success: function(events) {
12.                $('#calendar').fullCalendar('refetchEvents');
13.                $('#calendar').fullCalendar('rerenderEvents');
14.             }
15.         });
16.     }
```

Für den AJAX-Call müssen einige Informationen bereitgestellt werden. Das Datenformat über welches die Daten ausgetauscht werden ist JSON. Es muss die URL angegeben werden sowie die Informationen die zu einem Event-Objekt mitgeliefert werden. Ist der Speichervorgang erfolgreich, wird der Kalender neu geladen.

15. Anschauliches Beispiel

Um das Zusammenspiel der Dateien innerhalb von Django beispielhaft darzustellen, sei folgendes Szenario gegeben: Wir sind als User bei KOMPLETT angemeldet und wollen das Kontrollzentrum, also die Seite, um unser Profil und unsere Daten zu konfigurieren, öffnen. Weil wir als User angemeldet sind, gibt es im Hintergrund eine Instanz des LoginUser-Models mit unseren Daten. Dies ist das Model im Model-View-Controller Schema. Wenn der Link ../kontrollzentrum aufgerufen wird, wird in der urls.py das urlspattern von oben nach unten durchlaufen bis die URL, die auf das Kontrollzentrum verweist, gefunden wird. Diese URL verweist auf eine Funktion „control“ in der Datei views.py. Die views.py ist die Datei, die kontrolliert welche Inhalte für eine aufgerufene URL generiert und an das HTML-Template gesendet werden. Hier wird auch kontrolliert, welche HTML-Seite überhaupt dargestellt wird.

Die Methode „control“ ruft nun die Methode render auf, übergibt dieser Methode die entsprechende HTML-Datei und eine Info darüber, ob der User als Einzelspieler oder als Verein angemeldet ist. Dann wird die HTML-Seite mit den Informationen an den Browser gesendet. In der HTML-Seite wird überprüft, als welche Userart der User eingeloggt ist und das Kontrollzentrum mit den entsprechenden Funktionen angezeigt. Im Model-View-Controller Schema wird also die View-Funktionalität, hier von den HTML-Templates, implementiert.
Die views.py ist der Controller der die Navigation „lenkt“. Hier das Schaubild, beginnend mit dem LoginUser-Model:

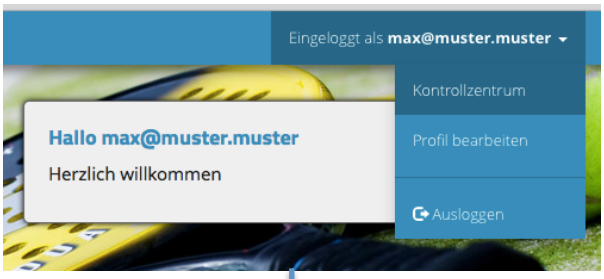
LoginUser-Model

```
# CustomUser
# Login
class LoginUser(AbstractBaseUser):
    login_id = models.AutoField(primary_key=True)
    email = models.EmailField(max_length=254, blank=False, unique=True)
    plz = models.PositiveIntegerField(blank=True, null=True)
    city = models.CharField(max_length=100, blank=True, null=True)
    is_active = models.BooleanField(default=False)
    is_admin = models.BooleanField(default=False)
    is_superuser = models.BooleanField(default=False)
    date_joined = models.DateTimeField(default=datetime.now)
    confirmation_code = models.CharField(default=generate_confirmation_code, max_length=160)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    @property
    def user_type(self):
        try:
            SinglePlayer.objects.get(pk=self.login_id)
            return UserType.single_player
        except ObjectDoesNotExist:
            try:
                TennisClub.objects.get(pk=self.login_id)
                return UserType.tennis_club
            except ObjectDoesNotExist:
                try:
                    LoginUser.objects.get(pk=self.login_id)
                    return UserType.admin
                except ObjectDoesNotExist:
                    return UserType.unknown
```

Darstellung des inneren Datenflusses der Applikation:



urls.py

```
url(r'^$', views.index, name='index'),
url(r'logout/', views.logout_user, name='logout'),
url(r'kontrollzentrum/$', views.control, name='control'),
...
```

views.py

```
@login_required
def control(request):
    return render(request, 'homepage/kontrollzentrum.html', {'user_type': request.user.user_type})
```

kontrollzentrum.html

```
<div class="content">
    <div class="container">
        <div class="col-lg-12">
            <p>Willkommen in Ihrem Kontrollzentrum.</p>
            {% if user_type == 'sp' %}{# single_player #}
                <p>
                    <a class="btn btn-primary" href="{% url 'homepage:profil' %}">Profil bearbeiten</a>
                </p>
            {% elif user_type == 'tc' %}{# tennis_club #}
                <p>
                    <a class="btn btn-primary" href="{% url 'homepage:tc_create' %}">Tennisplatz anlegen</a>
                    <a class="btn btn-primary" href="{% url 'homepage:court_list' %}">Tennisplätze bearbeiten</a>
                    <a class="btn btn-primary" href="{% url 'homepage:teams_add' %}">Mannschaft anlegen</a>
                    <a class="btn btn-primary" href="{% url 'homepage:teams_list' %}">Mannschaften bearbeiten</a>
                </p>
                <p>
                    <a class="btn btn-primary" href="{% url 'homepage:profil' %}">Profil bearbeiten</a>
                    <a class="btn btn-primary" href="{% url 'homepage:season_change' %}">Saisondaten bearbeiten</a>
                </p>
            {% else %}{# admin #}
                <h1>Hallo Adminuser</h1>
            {% endif %}
        </div>
    </div>
</div>
```

16. ER-Modell

Folgend das Entity-Relation-Modell von KOMPLETT: