Christian Steinmann (722503)
Markus Danilow (722451)

29. November 2015

# Webentwicklung: Das Gästebuch

*Verteilte Systeme*

## Die Aufgabenstellung

Im Rahmen des Praktikums der Veranstaltung *Verteilte Systeme* des Studienganges *Medien- und Kommunikationsinformatik* der Hochschule Reutlingen soll ein Gästebuch entwickelt werden, welches die Möglichkeit bietet, Beiträge zu lesen und zu verfassen.
Um die verfassten Beiträge dauerhaft speichern zu können, soll eine MySQL-Datenbank verwendet, der Server in PHP implementiert und das Frontend unter Verwendung von HTML, JavaScript und CSS realisiert werden. Die Kommunikation zwischen Client und Server soll dabei basierend auf AJAX im JSON-Format erfolgen.

## Das Konzept

Dieses Gästebuch stellt eine sogenannte *Single-Page Applikation* dar. Die gesamte Navigation sowie sämtliche Aktionen laufen dabei innerhalb der Seite ab, ohne dass ein erneutes Laden der Seite erforderlich ist. Gesamt setzt sich die Seite aus drei Hauptelementen zusammen: Für die Navigation zwischen den einzelnen Seiteninhalten existiert eine Navigationsleiste am linken Rand. Die verschiedenen Inhalte werden im Hauptbereich in der Mitte der Seite dargestellt. Für alle weiteren Optionen und Aktionen wie beispielsweise die Registrierung oder die Anmeldung gibt es die sogenannte Sidebar, die sich vom rechten Rand der Seite her öffnet. Der Fokus bei diesem Entwurf liegt dabei auf der Benutzerfreundlichkeit: Es sollen immer nur die Inhalte angezeigt werden, die für den Benutzer gerade im Moment von Interesse sind - alles andere wird konsequent ausgeblendet!
Des Weiteren verfügt das Gästebuch über zwei unterschiedliche Bereiche. Im öffentlichen Bereich kann jeder Benutzer anonym unter Verwendung eines temporären Benutzernamens neue Beiträge verfassen. Beiträge in diesem Bereich können von jedem gelesen werden, der die Seite aufruft. Zum privaten Bereich haben dagegen lediglich registrierte Mitglieder Zugang. Beiträge in diesem Bereich können von nicht angemeldeten Benutzern nicht eingesehen werden. Für die Registrierung werden ein noch nicht verwendeter Benutzername sowie ein Passwort benötigt.

## Die Datenbank

Die Datenbank besteht aus zwei Tabellen: Die Tabelle *users* beinhaltet alle Informationen über registrierte Mitglieder, wohingegen *content* der Speicherung aller verfassten Beiträge dient. Dabei werden sowohl Beiträge aus dem öffentlichen als auch solche aus dem privaten Bereich in dieser Tabelle abgelegt.

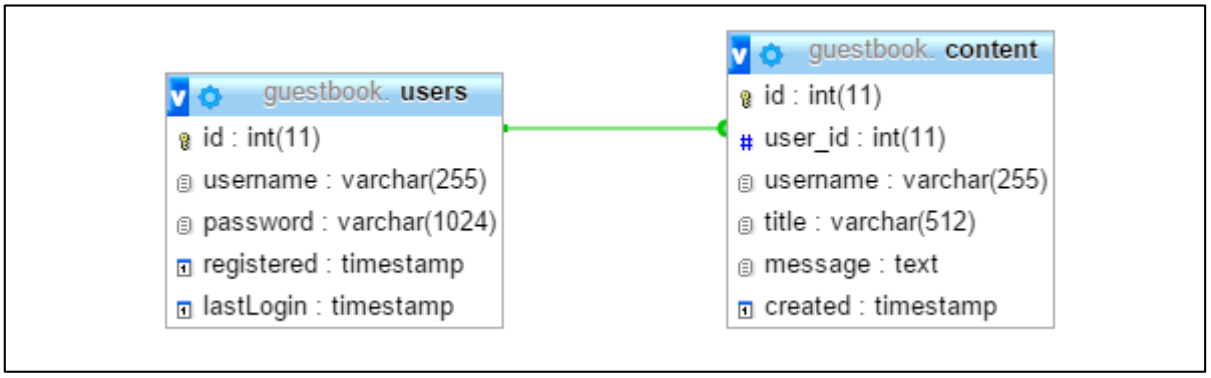


Abbildung 1: Das ER-Diagramm der Datenbank

Die Unterscheidung, welchem Bereich ein Beitrag angehört, geschieht über die angehängten Benutzerinformationen. Da private Beiträge nur von registrierten Benutzern verfasst werden können, wird bei diesen eine gültige *User ID* in der Datenbank hinterlegt. Bei öffentlichen Beiträgen kann keine User ID gesetzt werden, da diese von anonymen Benutzern erstellt werden. Daher wird hier direkt der temporäre Benutzername des Erstellers gespeichert, die User ID wird auf den Wert *NULL* gesetzt.

## Der Server

Die Dokumentation des Servers gliedert sich in die beiden Abschnitte *Datenbankanbindung* und *Schnittstellen* (Interfaces). Die nachfolgende Abbildung illustriert die grundlegende Architektur des Servers.

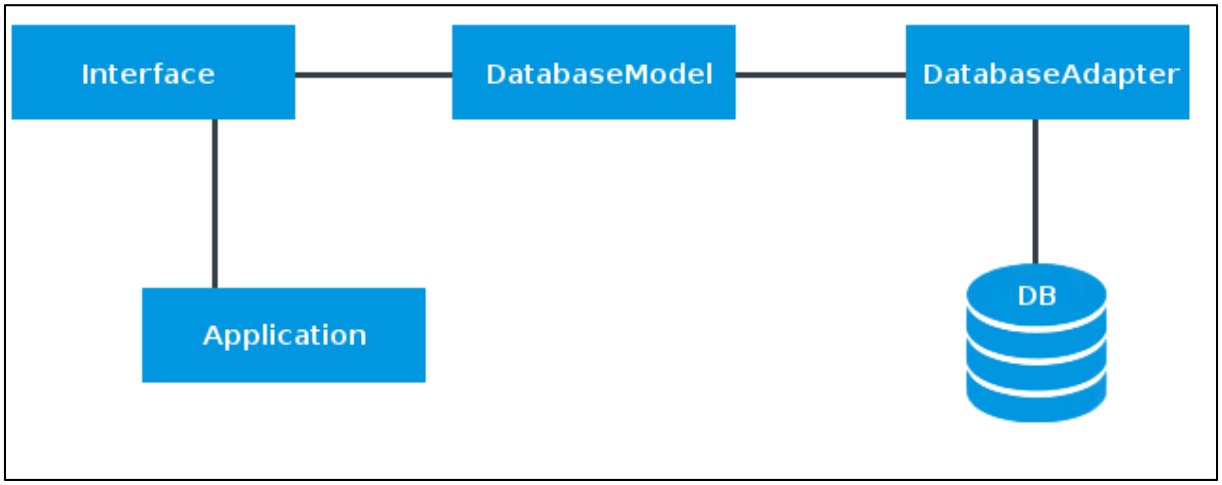


Abbildung 2: Die schematische Architektur des Servers

### Datenbankanbindung

Der *DatabaseAdapter* dient der Anbindung an die Datenbank und basiert auf den *PHP Data Objects* (PDO). Anfragen an die Datenbank werden dabei über die Methode *exec()* in Form von *Prepared Statements* gesendet. Falls es sich um ein *SELECT* handelt, werden die resultierenden Zeilen aus der Datenbank in die entsprechende Klasse konvertiert, die als Parameter spezifiziert werden kann. Die nachfolgende Abbildung zeigt die Methoden zum Verbindungsaufbau und -abbau sowie die Methode für die Ausführung von Datenbankanfragen.

```php
private function connect(){
    $this->connection = new PDO("mysql:host=".self::$DB_SERVER.";dbname=".self::$DB_NAME.";charset=utf8", self::$DB_USER, self::$DB_PASSWORD);
    $this->connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}

private function disconnect(){
    $this->connection = null ;
}

public function exec($query = "", $params = null, $clazz = null){
    if(!isset($params)){
        $params = array();
    }
    $producesResult = strpos($query, "SELECT") > -1;
    $isClazzDefined = isset($clazz);
    $this->connect();
    $statement = $this->connection->prepare($query);
    $statement->execute($params);
    $result = array();
    if($producesResult){
        if($isClazzDefined){
            $result = $statement->fetchAll(PDO::FETCH_CLASS, $clazz);
        }else{
            $result = $statement->fetchAll();
        }
    }
    $this->disconnect();
    return $result;
}
```

**Abbildung 3: Der DatabaseAdapter verbindet das Backend mit der Datenbank**

Das *DatabaseModel* bildet die Daten aus der Datenbank in objekt-orientierter Form ab und dient als Grundlage für die Implementierung aller weiteren Datenmodelle wie zum Beispiel das *UserModel*. Ein *Model* verfügt für die Kommunikation mit der Datenbank über eine statische Instanz des DatabaseAdapters. Außerdem existieren Methoden zum Speichern und Aktualisieren des Models in der Datenbank. Beide Prozesse funktionieren dabei auf Grundlage von *Reflection*. Hierdurch kann auch auf Attribute, die erst dynamisch zur Laufzeit gesetzt werden, zugegriffen und diese so in der Datenbank gespeichert werden. Die nachfolgende Abbildung gibt einen Überblick über die eben beschriebenen Methoden.

```php
/**
 * saves a new model to the database
 */
public function save(){
    $clazz = $this->getClass();
    $properties = $this->getProperties();
    $nameArr = implode(',', $properties['names']);
    $placeholderArr = implode(',', $properties['placeholders']);
    $query = "INSERT INTO ".$clazz::$tablename." (".$nameArr.") VALUES (".$placeholderArr.")" ;
    self::getDbAdapter()->exec($query, $properties['values'], $clazz);
    return true ;
}

/**
 * updates an existing model
 */
public function update(){
    $clazz = $this->getClass();
    $properties = $this->getProperties();
    $q_set = "SET ";
    for($i = 0; $i < sizeof($properties['names']); $i++){
        $q_set .= $properties['names'][$i]."=".$properties['placeholders'][$i];
        if($i < sizeof($properties['names'])-1){
            $q_set .= ', ' ;
        }
    }
    $query = "UPDATE ".$clazz::$tablename." ".$q_set." WHERE id=".$this->getID();
    self::getDbAdapter()->exec($query, $properties['values'], $clazz);
    return true ;
}

/**
 * @return array
 */
private function getProperties(){
    $reflect = new ReflectionObject($this);
    $props = $reflect->getProperties(ReflectionProperty::IS_PUBLIC | ReflectionProperty::IS_PROTECTED | ReflectionProperty::IS_PRIVATE);
    $tNames = array();
    $tValues = array();
    $tValuePlaceholder = array();
    foreach($props as $prop){
        $prop->setAccessible(true);
        if(!$prop->isStatic() && $prop->getName() !== 'id' && (!preg_match('/_/', $prop->getName()) || strpos($prop->getName(), '_') > 0)) {
            array_push($tNames,  $prop->getName());
            array_push($tValues, $prop->getValue($this));
            array_push($tValuePlaceholder, '?');
        }
    }
    return array("names" => $tNames, "values" => $tValues, "placeholders" => $tValuePlaceholder);
}
```

**Abbildung 4: Das DatabaseModel bildet Daten aus der Datenbank in objekt-orientierter Form ab**

Alle weiteren Models erben vom DatabaseModel und nutzen die Methoden dieser Klasse. Darüber hinaus verfügt jedes Model über individuelle Getter und Setter für die einzelnen model-spezifischen Attribute.

## Schnittstellen

Die Schnittstellen (Interfaces) dienen als Kommunikationspunkt zwischen Server und Client. Entsprechend der HTTP-Methode, den übergebenen Parametern und deren Gültigkeit werden die Anfragen des Frontends hier verarbeitet und Antworten im JSON-Format zurückgeschickt. Die nachfolgende Abbildung illustriert diese Funktionsweise anhand der Anmelde-Schnittstelle.

```php
require_once '../../initApplication.php' ;

if(isset($_POST['username']) && isset($_POST['password'])){

    $username = $_POST['username'] ;
    $password = $_POST['password'] ;

    if(UserModel::userExists($username)){
        $user = UserModel::getUserByName($username) ;
        if(PassHash::check_password($user->getPassword(), $password)){
            $user->setLastLogin(date('Y-m-d G:i:s'));
            $user->update();
            CustomSessionHandler::bindNewSessionParam(USER, $user->getUsername());
            Core::sendHTTPResponse(200, "Willkommen, ".$username.".");
        }else{
            Core::sendHTTPResponse(400, "Bitte gib Dein korrektes Passwort ein.");
        }
    }else{
        Core::sendHTTPResponse(400, "Es existiert kein Benutzer mit diesem Namen.") ;
    }

}else{
    Core::sendHTTPResponse(400, "Bitte gib sowohl Deinen Benutzernamen als auch Dein Passwort ein.") ;
}
```

Abbildung 5: Die Schittstelle zur Anmeldung eines Benutzers im Backend

Zunächst wird das *initApplication*-Skript eingebunden. Dieses wird in jeder Schnittstelle benötigt, um die *Application* und deren integrierten *Autoloader* nutzen zu können. Auf diese Weise muss ab diesem Zeitpunkt nicht mehr jedes verwendete Skript händisch eingebunden werden. Anschließend wird geprüft, ob es sich um die korrekte HTTP-Methode handelt und die erforderlichen Parameter alle gesetzt sind. Im Falle des Anmeldevorganges muss es sich um eine Anfrage via HTTP-Post handeln. Es werden der Benutzername und das Passwort benötigt. Sind diese korrekt eingegangen, wird überprüft, ob ein Benutzer mit dem angegebenen Benutzernamen existiert. Dies geschieht über den Aufruf der dafür zuständigen statischen Methode des *UserModels*. Existiert kein Benutzer mit diesem Namen, wird - wie in jedem anderen Fehlerfall - eine Antwort mit dem *HTTP-Statuscode* 400 (client-seitiger Fehler, zum Beispiel durch fehlerhafte Eingabe) und einer optionalen Fehlermeldung an den Client zurückgesendet. Ist der Benutzer dem System bekannt, wird das dazugehörige UserModel geladen und das eingegebene Passwort mit dem in der Datenbank hinterlegten verglichen. Selbstverständlich erfolgt dies in verschlüsselter Form. Zuständig für die Verschlüsselung und die Prüfung verschlüsselter Passwörter ist die statische Klasse *PassHash*. War die Passworteingabe ebenfalls korrekt, so hat sich der Benutzer erfolgreich angemeldet. Der Zeitpunkt der letzten Anmeldung wird in der Datenbank aktualisiert und die *Session* des Benutzers aktiviert. Im letzten Schritt wird eine Willkommensnachricht mit dem Statuscode 200 an den Client gesendet. Alle anderen Schnittstellen funktionieren auf dieselbe Weise.
Nachfolgend wird noch kurz die Methode illustriert, die die HTTP-Antworten im JSON-Format an den Client sendet. Diese befindet sich im Modul *Core*.

```php
public static function convertToJSON($array){
    return json_encode($array) ;
}

public static function sendHTTPResponse($statusCode, $message = "", $data = null){
    header($_SERVER['SERVER_PROTOCOL'].' '.$statusCode.' '.$message, true, $statusCode);
    print self::convertToJSON(array('statusCode' => $statusCode, 'message' => $message, 'data' => $data));
}
```

Abbildung 6: Senden einer HTTP-Antwort an den Client

## Der Client

Wie der Server gliedert sich auch der Client in mehrere Komponenten: So gibt es client-seitige Schnittstellen (Interfaces), die zur Kommunikation über HTTP-Methoden mit dem Server dienen, Skripte, die die Navigation auf der Seite sowie Animationen regeln und zuletzt HTML-Dateien, die über AJAX in die entsprechenden Bereiche der Seite eingebunden und unter Verwendung von CSS grafisch gestaltet werden. Die gesamte in JavaScript realisierte Funktionalität wird dabei durch den Einsatz von *jQuery* unterstützt.

### Schnittstellen

Auf Seite des Clients existieren Schnittstellen für *HTTP-Get* und *HTTP-Post*. Diese rufen über AJAX eine bestimmte URL auf und senden optional ein Datenpaket im JSON-Format an den Server. Bei der URL handelt es sich um den Pfad zur entsprechenden Schnittstelle des Servers. Außerdem kann über die Parameter jeweils eine Funktion für den Erfolgsfall und den Fehlerfall der asynchronen Anfrage als sogenannte *Callback*-Funktion definiert werden. Das Ergebnis des Servers trifft in beiden Fällen wiederum im JSON-Format ein und kann anschließend in den Callback-Methoden interpretiert und weiterverarbeitet werden. Die nachfolgende Abbildung zeigt die Schnittstelle des Client zur Kommunikation mit dem Server über HTTP-Post. Das Interface für HTTP-Get funktioniert auf dieselbe Weise. Lediglich das Attribut *method* beinhaltet anstatt *"post"* den Wert *"get"*.

```
function PostInterface(){ }

PostInterface.execute = function(url, data, successCallback, errorCallback){
    $.ajax({
        url: url,
        method: 'post',
        data: data,
        async: true,
        dataType: 'json',
        success: function(sData){
            if(successCallback){
                successCallback(sData);
            }
        },
        error: function(eData){
            if(errorCallback){
                errorCallback(eData);
            }
        }
    })
};
```

Abbildung 7: Senden einer asynchronen HTTP-Anfrage an den Server

Diese Schnittstellen können dann, wie in der nachfolgenden Abbildung zu sehen ist, beispielsweise für die Anmeldung eines Benutzers verwendet werden. In der Methode *performLogin()* werden zunächst der Benutzername und das Passwort aus dem Anmeldeformular ausgelesen und auf Korrektheit bzw. Vollständigkeit geprüft. Ist die Prüfung erfolgreich, kann der Anmeldeprozess über *doLogin()* und HTTP-Post gestartet werden. Die Methode *execute()* des PostInterface erhält dabei zwei Callback-Funktionen als Parameter, die auf die Antwort des Servers entsprechend reagieren und passende Meldungen in der Benutzeroberfläche des Client ausgeben.

```
function performLogin(){
    var username = $('#login_username').val();
    var password = $('#login_password').val();
    $('#login_error_username').addClass('invisible');
    $('#login_error_password').addClass('invisible');
    var check = checkValues([username, password]) ;
    if(check.correct){
        doLogin(username, password);
    }else{
        handleInvalidLogin(check);
    }
}

function doLogin(username, password){
    password = CryptoJS.MD5(password).toString();
    PostInterface.execute(urlAPI+'user/login.php', {username: username, password: password}, loginSuccess, loginError);
}
```

Abbildung 8: client-seitiger Anmeldvorgang eines Benutzers

Neben den Interfaces existieren noch andere JavaScript-Dateien, wie beispielsweise die für die Navigation. Die in der nachfolgenden Abbildung dargestellte Funktion weist jedem Link auf der Seite ein *onClick*-Ereignis zu. Je nach Typ des Links wird das verlinkte HTML-Dokument entweder in den Hauptbereich in der Mitte der Seite oder in die Sidebar eingebunden.

```
function bindClickEvent() {
    $('body').on('click', 'a',  function (e) {
        e.preventDefault();
        if (!$(this).hasClass('active') && !$(this).hasClass('notActivated')) {
            var scope = this;
            closeSidebar(function(){
                if ($(scope).hasClass('special')) {
                    $.each($('body').find('a'), function (index, elem) {
                        $(elem).removeClass('active');
                    });
                    $('.contentArea').load($(scope).attr('href'));
                    $(scope).addClass('active');
                } else {
                    openSidebar(function () {
                        $('.sidebarContent').load($(scope).attr('href'));
                    });
                }
            });
        }
        return;
    });
}
```

**Abbildung 9: Zuweisung eines einheitlichen *onClick*-Ereignisses für alle Links der Seite. Hierdurch wird der Single-Page-Charakter der Seite gewährleistet.**

## HTML Templates

Die HTML-Dateien gliedern sich ihrerseits wiederum in *components*, *sidebar* und *content*. *Components* bilden das Grundgerüst der Seite. Hierzu zählen unter anderem die Sidebar als Element selbst, das Menü am linken Rand sowie der Header, der alle CSS- und JavaScript-Dateien einbindet. Bei *content* handelt es sich um alle Unterseiten, die direkt im Hauptbereich in der Seite eingebunden werden, so wie beispielsweise diese Dokumentationsseite. Als einfaches Beispiel für ein solches HTML Template wird in der nachfolgenden Abbildung das Anmeldeformular gezeigt, das bei Klick auf diverse Links in die Sidebar eingebunden wird.

```
<div class="absoluteElement fullPageContainer">
    <div class="contentContainer">
        <h2>Anmelden</h2>
        <p id="login_message">Bitte gib Deine Anmeldedaten in das Formular ein, um Dich anzumelden.</p>
        <p>Falls Du noch kein Konto hast, kannst Du Dich <a href="templates/sidebar/user/register.html">hier</a> registrieren.</p>
        <form>
            <table>
                <tr>
                    <td class="rightTableCol">
                        <input type="text" id="login_username" value="" placeholder="benutzername"/>
                    </td>
                    <td class="error invisible" id="login_error_username">
                        Bitte gib Deinen korrekten Benutzernamen ein.
                    </td>
                </tr>
                <tr>
                    <td class="rightTableCol">
                        <input type="password" id="login_password" value="" placeholder="passwort"/>
                    </td>
                    <td class="error invisible" id="login_error_password">
                        Bitte gib Dein korrektes Passwort ein.
                    </td>
                </tr>
                <tr>
                    <td class="rightTableCol">
                        <input type="button" id="login_submit" value="anmelden" onclick="performLogin();"/>
                    </td>
                    <td>

                    </td>
                </tr>
            </table>
        </form>
    </div>
</div>
```

**Abbildung 10: Das Anmeldeformular**

Das Formular setzt sich aus zwei Eingabefeldern und einem Button zusammen. Die Eingabefelder für Benutzername und Passwort verfügen jeweils über einen *Placeholder*, um dem Benutzer anzuzeigen, was in das jeweilige Feld einzutragen ist. Außerdem gehört zu jedem Eingabefeld ein standardmäßig unsichtbares Feld mit einer entsprechenden Fehlermeldung. Wurde die Anmeldung z.B. aufgrund einer fehlerhaften Eingabe des Benutzers nicht korrekt durchgeführt, werden die jeweiligen Fehlermeldungen in der entsprechenden JavaScript-Funktion eingeblendet. Bei Klick auf den *"anmelden"*-Button wird letztendlich die *performLogin()*-Methode aufgerufen. Deren Funktionalität wurde im vorhergehenden Abschnitt bereits erläutert.

## Polling – Asynchrone Aktualisierung der Beiträge

Für das Laden und Anzeigen der Beiträge wird *Polling* verwendet. Die nachfolgende Abbildung illustriert das Prinzip, das sowohl für die privaten als die öffentlichen Beiträge verwendet wird.

```
<script>
    quitAllAsyncIntervals();
    registerOnServer();
    contentOffset = 0;
    contentInitialLoad = true ;
    /* fetch everything initially */
    loadPublicContent(Math.pow(2, 53) - 1);
    /* then set interval for periodically fetching */
    publicPollInterval = setInterval(function () {
        loadPublicContent(contentLimit);
    }, pollingIntervalTime);
</script>
```

Abbildung 11: Das Prinzip des Pollings in einem definierten Intervall für die Aktualisierung von Beiträgen

Zunächst werden alle asynchronen Ladefunktionen beendet, die eventuell noch im Hintergrund aktiv sind. Anschließend findet eine Registrierung des Clients auf dem Server statt. Tatsächlich handelt es sich dabei server-seitig nur um die Initialisierung eines neuen Session-Parameters, der dazu dient, dass tatsächlich nur die neusten und nicht etwa immer wieder alle Beiträge geladen werden. Zu Beginn werden alle Beiträge einmal geladen. Bei *Math.pow(2, 53) - 1* handelt es sich um den größtmöglichen Integer-Wert; dieser wird als Limit für die spätere server-seitige Datenbankabfrage verwendet. Somit kann angegeben werden, wie viele Beiträge maximal auf einmal geladen werden sollen. Ab diesem Zeitpunkt läuft die Ladefunktion in einem festen Intervall ab und aktualisiert die Beiträge im Abstand von *pollingIntervalTime* Millisekunden. Der blaue Balken am oberen Rand der Seite fungiert dabei als eine Art "virtuelle Sanduhr": Ist der Balken durchgelaufen, werden die Inhalte aktualisiert.

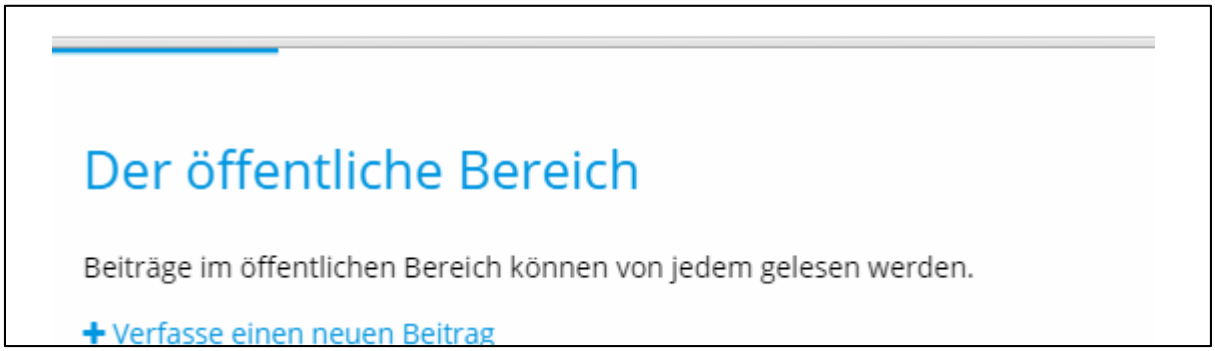


Abbildung 12: Der Polling-Fortschrittsbalken: eine virtuelle Sanduhr

Sind neue Beiträge verfügbar, so werden diese via jQuery in das dafür vorgesehene Template *contentBox.html* geladen und das daraus resultierende jQuery-Objekt durch Aufruf der Methode *prepend()* den anderen Beiträgen vorangestellt. Eine Animation der Transparenz und der Umrandung sowie das Erscheinen des Popup-Fensters am oberen Rand weisen den Benutzer visuell darauf hin, dass neue Beiträge geladen wurden.

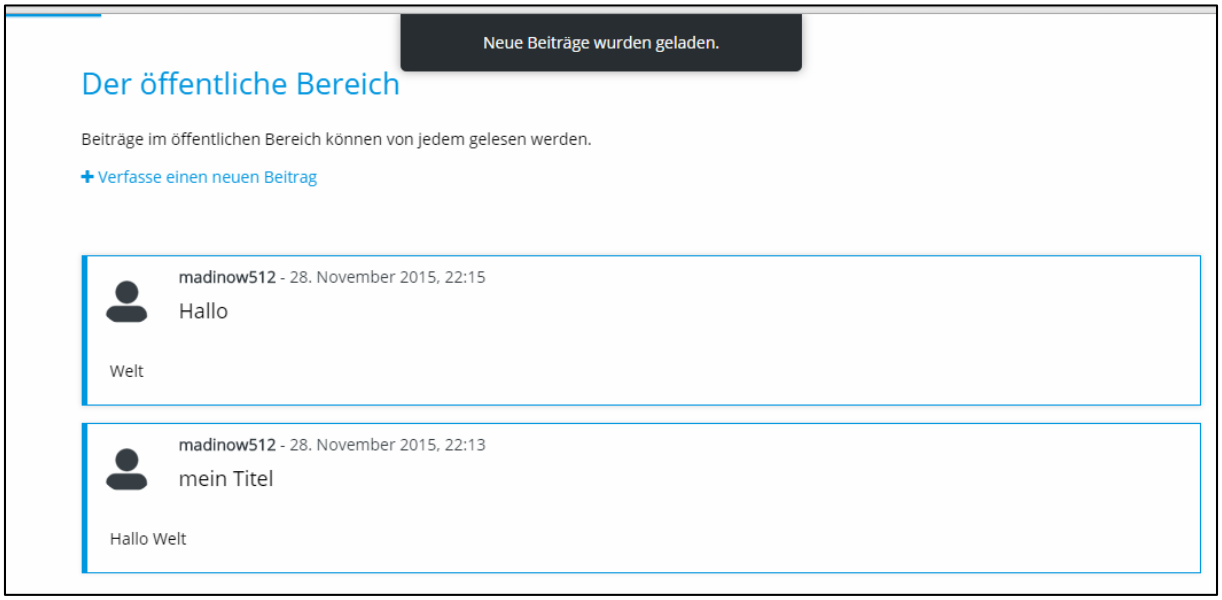


**Abbildung 13: Visuelle Hervorhebung neuer Beiträge**