INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

A Social Platform for Bipartite Student-Lecturer Ranking

Julien Oster

Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Christoph Wieser
Abgabetermin:	28. November 2011

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 28. November 2011

....

Julien Oster

Abstract

Conventional lecturer rating platforms base their rankings on students' ratings alone, without taking the students' performances into account. The information about students' performances, however, is readily available, since lecturers rate students by grading them in examinations. Bringing together lecturer-given grades and student-given ratings leads to a bipartite rating graph that can be used with ranking techniques which rank students and lecturers alike, and feed back student and lecturer ranks to govern the respective influence of their ratings.

We propose an algorithm called *eval²Rank*, based on PageRank and developed for bipartite ranking situations, and adapt this general eval²Rank algorithm to the specific intricacies of Student-Lecturer ranking. The proposed ranking is then an eigenvector centrality measure providing the mutual reinforcement typical of such measures. eval²Rank differs from PageRank in the inclusion of (normalized) edge weights and a modification of how random leaps are applied, as an adaption to the bipartite input graph.

We further present a fully functional Web Application, `eval`, which provides an intuitive Web Platform for the rating and ranking of lecturers by students, using student grade transcripts to feed in the students' performances. The Web Application is live-deployed as a prototype, and the gathered data used for a brief analysis of the algorithm's validity.

Acknowledgments

First, I would like to thank my advisor, Christoph Wieser, who gave greatly valuable guidance through the whole process of writing this thesis and who, along with all his colleagues at the Teaching and Research Unit for Programming and Modelling Languages, provided a friendly and open environment to work in.

I would also particularly like to thank Prof. Dr. François Bry, who had the idea for the algorithm and the whole platform, and who followed the whole project with great enthusiasm and lots of helpful advice.

Thanks also to my family, who supported me the entire time, not only during this thesis, but throughout my whole studies.

And of course, thanks to all my friends and fellow students, whose invaluable company in- and outside of the university buildings was always greatly appreciated, and who made this stage of my life the formidable time that it was.

Julien Oster

Contents

1. Introduction

With the advent of the so-called *Web 2.0*, perception towards the Web shifted from static or semi-static content governed by a few to strong forms of user participation and user-generated content. One such form of participation is the act of *rating*, i.e. sharing one's own opinion with the community as a data point on a more or less rigidly defined scale, possibly also accompanied by free-form reviews.

What exactly is being rated by the user varies from platform to platform, though the spectrum on which rating is applicable seems virtually without limits. One of the earliest examples is eBay [11], where sellers and buyers rate each other with respect to their recent transaction. One of the most recent examples, on the other hand, is the social interaction platform Facebook, where a comparably simple form of rating, the so-called *Like button* [15], is indeed already applicable to practically any form of content.

In this thesis, we focus on ratings in the context given between students and lecturers of universities.[1] By the time of this project's conception, there already are numerous websites in existence which allow students to publicly rate lecturers, often targeted at the student audience of specific countries. Among those websites are, for example, *ratemyprofessors.com* for universities and colleges in the United States of America, and *meinprof.de* for universities in Germany. These websites give students the possibility to rate lecturers either directly or via the lectures they have given, and create various rankings using those ratings.

This gives students and lecturers an important tool that was not readily available before the Web's aforementioned forms of strong user participation evolved: Based on other students' perception, students are now able to make informed decisions on which lectures –or even which universities– to attend, and lecturers are able to improve their teachings with the aid of broad feedback, especially in relation to other lecturers in the respective ranking, and not just with the narrow view of single-lecture evaluations.

To the best of our knowledge, however, none of the currently existing websites seem to take the *other* possible direction of rating into account: the rating from lecturer to student. Recall that with eBay, for example, buyers and sellers rate each other alike. With lecturer rating websites, the ratings are one-sided, and ratings originating from lecturers are not part of any evaluation. Some of the websites employ certain "safe-guards", for instance by giving lecturers the ability to write rebuttals to their received ratings and reviews or by allowing users to verify their status as an enrolled student of the respective university, and even assigning a higher weight to verified students when calculation lecturer rankings. Nevertheless, the situation remains highly asymmetric.

In result, it is questionable how expressive the given ratings and especially the resulting rankings really are. It is entirely conceivable that a small group of students with low academic performance gives significantly bad ratings to a lecturer just because they did not reach

[1] Although any of the approaches introduced in this thesis can be easily adapted to similar situations, e.g. students and teachers at high schools.

the lecturer's expectations on students in general, even with the lectures and materials themselves being rather good, and with other, better performing students having no such problems. However, if none or too few of the better performing students also participate in the rating of that particular lecturer, the overall image of the lecturer as reflected into the ranking will be bad. Similarly, some students may rate lectures highly just due to particularly easy exams, when it is clear to other students that the lectures' content itself appears to be sub-par. Hence, it might be fairer and, above all, more conclusive in its results to allow the *students'* performance to play a key role in the calculation of any rankings themselves.

However, a rating from lecturer to students trivially exists already: the students are all graded by the lecturers through examinations, which are usually associated to the students' attended lectures.

The idea is thus to unify both the grades given by lecturers and the ratings given by students into the notion of mutual ratings, and to use either aspect as information when generating rankings. Intuitively, the result is that not every student's rating will be weighted equally, instead students are ranked as well and their respective rank should basically govern the influence given to their ratings. On the other hand, though, the same should apply for lecturers: Vice versa, lecturers which are ranked as "good" should play a higher role in the ranking of students overall. Taking the mutual influences of either partition (formed by students and lecturers respectively) together, a kind of feedback flow emerges which influences either rankings until they converge on what can be thought of as a mutually agreed ranking of both partitions.

If we model the whole system as a graph, using participants (students and lecturers alike) as nodes and ratings/gradings as edges, eigenvector centrality measures like the seminal PageRank algorithm [7] come to mind, as it uses exactly the aforementioned principle of flowing relevance. We can indeed use an algorithm derived from PageRank to create more expressive rankings, however with some modifications. These modifications are the adaption of the eigenvector calculation to bipartite network/graph situations, introduction of weights and a modification of how random jumps are incorporated for bipartite graphs.

Besides the need for an algorithm that implements this notion of a *bipartite ranking*, an approach to operate this algorithm in real-life needs to be developed as well in the form of a fully fledged Web-platform, but with a focus on feasibility as well. For example, if the platform requires all lecturers to manually input the students' grades, chances of sufficient participation are slim. Instead, we aim to make the platform as self-sustaining as possible, reducing the amount of necessary action for all participants to a minimum, sometimes even to nil. The existence of student transcripts, which represent official records of all student grades, plays a significant role in simplifying the elicitation of data. Students need only to agree to the import of their transcripts into the system and rate lecturers (via lectures) at will, while lecturers do not need to take any action at all.

The contribution of this thesis is thus the conception and elaboration of a bipartite ranking algorithm called eval²Rank and based on PageRank, as well as the development of a fully functional Web Application which uses eval²Rank. Initial live deployment of a first prototype of that Web Application is performed, followed by a brief analysis of the proposed algorithm using the data gathered with the prototype.

This thesis is organized as follows: after this introduction, we will first discuss related work. Subsequently, in chapter 3, we will fully develop and describe an algorithm for bipartite ranking in general. This is done by modifying the original PageRank algorithm with weights and by taking the graph's bipartiteness into consideration. In chapter 4, we will adapt the algorithm and its input to the actual Student-Lecturer-Ranking situation. Following this, chapter 5 describes the Web Application, called `eval`, that employs the fully developed algorithm, focusing on the user's perspective. Implementation of `eval` is covered in chapter 6, including an explanation of the used data model as well as the used frameworks, technologies and developed components. Chapter 7 describes deviations and extensions in the deployed prototype and chapter 8 follows up with a brief analysis of the algorithm given the data gathered with said prototype. Chapter 9 finally gives a conclusion and proposes future work.

Some emerging aspects and ideas are not covered within the scope of this thesis. Among them is a rigorous analysis of theoretical properties of the algorithm itself as well as an evaluation using appropriately crafted data. Also, this thesis only examines a bipartite approach, although tripartite or other multipartite approaches are conceivable, such as through a direct involvement of the rated lectures. Those and other considerations are left as possible directions for future work in this area.

2. Related Work

Newman [27] describes, in detail, the network centralities that our approach is based on: the basic *eigenvector centrality* proposed by Bonacich [6], the *Katz Centrality* [22] which first contains the addend of what is now commonly known as the "random jump" or "random leap" and finally the algorithm which in a way popularized such centralities in the field of computer science, *PageRank*.

Google's PageRank, the initial basis of the algorithm for rankings in bipartite graphs that we propose in this thesis, is first described in its present form by Brin and Page [7]. Due to the often large sizes of the graphs it is operating on, PageRank is most commonly computed using the Power Method, as described by e.g. Haveliwala et al. [17]. A good introduction to PageRank and its calculation is given in lectures by Bry et al. [8]. Another introduction to the linear algebra behind PageRank is provided by Bryan and Leise [9]. Deeper insight into its properties has been given by Bianchini et al. [4] and furthered by Langville and Meyer [25].

However, PageRank is described and commonly used on graphs where all nodes are considered to be of the same kind, e.g. webpages without any further classification. In our method, we are interested in a kind of PageRank on bipartite graphs, where nodes are of two different kinds (i.e. students and lecturers) and form the partitions respectively.

While there is plenty of literature that discusses PageRank or similar approaches with respect to bipartite graphs, most of the literature seems to focus on *relevance search* according to a given node, as opposed to absolute rankings of all nodes of the two respective partitions. For example, Sun et al. [30] propose algorithms for relevance search and anomaly detection by using random walks, the principle behind PageRank. As possible applications the paper states similarity and anomaly detections in P2P systems (files and peers), stock markets (traders and stocks) and research publications (conferences and authors). Related, Bauckhage [2] proposes an algorithm to tag images by using PageRank over bipartite graphs. The nodes in the input for this algorithm are prototypical objects and image features, respectively.

These approaches are however based on being given a query node and then finding similar objects (or anomalies) with respect to that node. For our case, we are presently not interested in establishing similarity or neighborhoods, instead we strive for absolute rankings for both partitions respectively.

One algorithm that specifically does create such rankings over each partition of even tripartite graphs is *FolkRank*, as devised by Hotho et al. [20]. FolkRank creates rankings for so-called folksonomies, which are essentially resources collaboratively tagged by users. Rankings over the users, tags and resources are then created, using a modified PageRank algorithm. The underlying graphs are undirected rather than directed as in rating situations, though this does not affect how the actual computation is performed. While ranking, specific nodes (of any kind) can also be weighted higher, to produce rankings for specific *topics*.

This leads to another very important aspect, namely the *weights* themselves. Original PageRank is based on a binary relationship between any two nodes: there either exists an edge from one node to another, or there does not. This is not enough for our application: a rating between two participants is not just either there or not there, it is also denoted by a number in a specific scale (or multiple numbers, but not within the scope of this thesis).

There are numerous other approaches to introduce weights into PageRank, and we will name some of them as examples to explore the different motives for the introduction of weights and their various approaches. Aktas et al. [1] weight webpages by user interest profiles, exemplarily using the site's Internet domain in their paper. Yu et al. [32] add weights to reflect the age of citations, to later favor more recent citations over older ones, and Sun and Giles [31] follow a similar approach with the citations' popularities.

Some of these approaches can be modeled with node weights (such as FolkRank), some with edge weights. We are presently only interested in edge weights, as we do not impose any intrinsic weights on any participants.

Also note that not all approaches make an attempt to keep the column-stochastic property of the resulting transition matrix (explained in section 3.6), i.e. the weights may result in some columns having a sum not equal to 1. Theoretical implications aside, we want to avoid this primarily for the sake of interpretation, as discussed in section 3.7.

Of particular interest is a paper by Ding et al. [12], though, which keeps the aforementioned property by normalizing the columns to one. Our approach that emerges from generalization of multigraphs into simple but weighted graphs effectively leads to a form of edge weights that are equivalent to the added weights with column normalization proposed by Ding et al. [12]. The paper overall presents a more general approach in which even random leaps are imposed with differing weights (i.e. node weights), which are, as mentioned before, however not relevant for our conditions.

For the sake of our algorithm's interpretation and the validity of its imposed modifications, we will try to motivate and develop our approach "from scratch", i.e. with only the original PageRank as a starting point.

3. The Bipartite eval²Rank Algorithm

This chapter describes the eval²Rank Algorithm, which is based on PageRank [7]. To fit into PageRank, we will first model the mutual ratings as a graph. In this graph, students and lecturers are represented as nodes, and form two disjoint partitions respectively. We will then motivate the usage of a PageRank-based algorithm, and briefly describe the original PageRank. Following that, weights will be introduced to incorporate ratings. After an interpretation of the resulting transition matrix, we will modify PageRank's random leaps to become more sound with bipartite ranking conditions.

Note that while the terms "student" and "lecturers" are used throughout this chapter, on the one hand because the algorithm targets Student-Lecturer-Ranking and on the other hand because they provide a readily available verbal distinction between those two partitions, this chapter describes the algorithm in general and is easily adaptable to any other situation where bipartite ranking is desired, e.g. Buyer-Seller ranking. Considerations which are highly specific to Student-Lecturer ranking are instead detailed in the next chapter.

3.1. The Rating Graph

The grading of students by lecturers, as well as the rating of lecturers by students, can easily be represented as a graph. In such a graph, both students and lecturers are represented as nodes, while a grading or rating is denoted by an edge between two nodes in the appropriate direction, making the graph a *directed* graph. Because lecturers only grade students, and students only rate lecturers, the graph is also *bipartite*.

(In the following, we will not always verbally distinguish between *grading* and *rating*, and may use the word *rating* for both cases if the meaning remains clear, possibly specifying the edge's direction if relevant.)

In its entirety, the graph that our algorithm should operate on is a graph $G = (S \cup L, E)$ in which the following properties hold:

1. Since the graph is *bipartite*, the sets for students S and lecturers L, forming the graph's set of vertices $S \cup L$, are disjoint, i.e. $S \cap L = \emptyset$.

2. The graph is a *directed graph*, i.e. each edge in its set of edges E is an ordered pair. Because it is also bipartite, it further follows that only edges connecting nodes of the two disjoint sets are allowed, i.e. $E \subseteq (S \times L) \cup (L \times S)$.

3. The edges have positive real weights, i.e. there is a function $w : E \to \mathbb{R}^+$ that assigns a weight to each existing edge. For the sake of simplicity, we will often denote the weights as $w_1, w_2, \ldots, w_{|E|}$, where w_i constitutes the weight of edge e_i (the enumeration of the edges is, unless otherwise noted, arbitrary). Edges with weight 0 are not allowed, instead non-existing edges are sometimes regarded as weighted with 0.

3.2. Rating Graph Examples

To illustrate our approach, we present two rating graphs that will serve as examples throughout our discussion.

We also provide modified adjacency matrices for these graphs. Common adjacency matrices of unweighted directed graphs are quadratic matrices $A = [a_{i,j}]_{n\times n}$ where each entry $a_{i,j}$ is 1 if there is an edge from node i to node j and 0 if not. For weighted graphs, the cells of their corresponding modified adjacency matrix directly contains the weight of its edges instead of 1s.

Note that PageRank is usually calculated using a column-stochastic matrix. With PageRank in undirected graphs, the adjacency matrix is symmetric and thus it is irrelevant whether we regard column- or row-stochastic matrices. Directed Graphs, however, are generally not represented by symmetric adjacency matrices. Since we want to keep the column-stochastic approach, we follow the definition given by Spielman [29], which, as stated in that lecture, is opposite of the expected definition, namely in the way that an edge from a node u to a node v is denoted by an entry in the *column* of u and the *row* of v, i.e. the entry $a_{v,u}$.

We will see that this transposition will keep the notion of normalizing a column vector with the respective participant's total rating output sound.

3.2.1. Non-bipartite Rating Graph

We first provide a simple example of a *non*-bipartite rating graph, which is better suited for illustration of aspects where bipartiteness is not relevant, as it is less constrained and allows for a more compact representation.

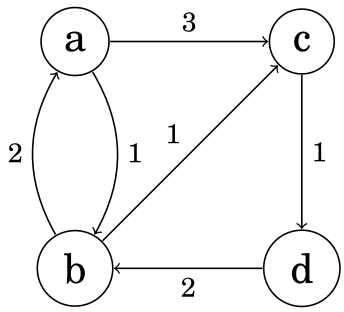


Figure 3.1.: A simple non-bipartite rating graph

$$\begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{pmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 3.2.: The modified adjacency matrix of graph 3.1

3.2.2. Bipartite Rating Graph

For aspects where bipartiteness plays a key role, we also provide simple bipartite rating graph, consisting of three students and two lecturers.

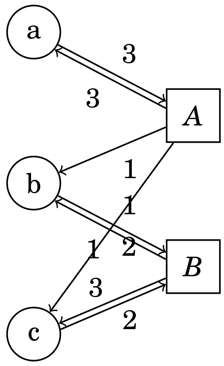


Figure 3.3.: A simple bipartite rating graph

$$
\begin{array}{c c}
 & \begin{array}{ccccc} a & b & c & A & B \end{array} \\
\begin{array}{c} a \\ b \\ c \\ A \\ B \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 3 & 0 \\
0 & 0 & 0 & 1 & 2 \\
0 & 0 & 0 & 1 & 3 \\
3 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 0 & 0
\end{pmatrix}
\end{array}
$$

Figure 3.4.: The modified adjacency matrix of graph 3.3

3.3. Motivation For Using Eigenvector Centrality À La PageRank

Our desire is to create a ranking of the nodes in this graph which, in its whole, is as unbiased as possible. The idea is that the rating of overall good students should have more influence on the overall ranking than the rating given out by bad students. However, the idea of a "good" student needs careful definition. An initial idea is that the scholar quality of a student is determined by the grades the student has been given. However, just like a student's rating should have a variable influence according to his grades, the grading *itself* should have variable influence on the student's relevance based on the lecturer's ratings. Intuitively, in our approach, every rating a node n has been given to should influence the relevance of the ratings that node n is imposing on another node m, while the relevance of node m's ratings should in turn be adjusted according to all ratings given towards node m, and so on. In that way, students that received a lot of good grades by lecturers with a high rank may themselves give out highly relevant ratings–there seems to be a consensus that such a student may be able to give out a valuable opinion.

Note that the crucial part is that this consensus is not just on part of the lecturers that awarded the student with high grades, but within the whole system, because only lecturers that themselves have achieved a high rank (in turn as a result of high ratings of highly ranked students) may influence the relevance of that student's rating in a significant way.

This is all very reminiscent of the *PageRank* algorithm, and indeed PageRank seems a good fit: nodes within a directed graph are to be ranked, whereby relevance of a node should flow to all nodes connected via an outgoing edge. Our goal is thus to adapt PageRank into a viable algorithm for our situation.

At first, we will set aside the fact that the graph we are dealing with is bipartite. While this will prompt a modification on the algorithm to allow for implications of this graph structure later on, we will first look at the nodes and their rank as if they weren't part of any specific group, i.e. we will first look at them as overall *participants* instead of *students* and *lecturers* respectively. Of course, this implies that the ratings given in both directions are on a somehow comparable scale. We will assume that this is the case, and later ensure it in our implementation.

3.4. Original PageRank

One comprehensive way of understanding how PageRank works is to look at the equation that defines the rank of any individual node p_i, recursively depending on the ranks of its incoming (= rating) nodes [7]:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in I(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{3.1}$$

whereby

- N is the total number of nodes,
- d is the so called *damping factor* ($0 < d < 1$),
- $I(p_i)$ is the indices of nodes that have an outgoing edge to p_i and
- $L(p_j)$ is the number of outgoing edges *from* p_j.

PageRank was originally conceived for the ranking of hypertextual Web pages, in which case $I(p_i)$ specifically denotes the set of pages that link to p_i and $L(p_j)$ the number of links to other pages on p_j.

The first part of the PageRank sum, $\frac{1-d}{N}$ is the so called *random leap* factor. Random leaps distribute relevance not along connected nodes, but, to a lesser extent, uniformly across *all* nodes. We will take an in-depth look at this part and how to handle it in our situation once we get to the intricacies of our graph's bipartitness. For now, we will concentrate on the factor $\sum_{p_j \in I(p_i)} \frac{PR(p_j)}{L(p_j)}$ in the PageRank formula.

This part asserts that the major[1] part of a node's rank is the sum of the ranks of all nodes that connect to it, each rank however divided by the number of outgoing edges its node has. Thus, a node's rank or relevance flows uniformly to all its successor nodes. While this fits one requirement of our ranking approach, namely that relevance of a participant should flow to its rated participants, it misses the crucial part of our endeavor: namely that participants aren't just simply connected to other participants in a binary "*is* or *is not*" relation, but that

[1] under usual choices for d, such as 0.85 [7]

the pivotal significance of this connection is the actual rating, which it carries between those participants. We achieve this by incorporating those ratings as *weights* within the modified PageRank approach.

3.5. Weighted PageRank

There are several efforts to introduce weights into PageRank. For example, Aktas et al. [1] add node weights for Web search personalization, Yu et al. [32] edge weights to add a temporal dimension to citation networks (newer citations are favored).

We present an approach that emerges naturally from the notion of ratings between participants. This approach has the benefit that, besides making sense of the bipartiteness of a rating graph, it keeps the resulting transition matrix a left-stochastic matrix as in the original unweighted PageRank, while still allowing arbitrary edge weights. Therefore, the transition matrix's entries may be interpreted as probabilities, which allows for an intuitive understanding of the validity of our approach.

An equivalent form of those edge weights is presented by Ding et al. [12], who however arrive there by directly normalizing the columns of the symmetric matrix of an *undirected* graph, which therefore becomes asymmetric.

3.5.1. Using Multigraphs To Model Integer Weights

Let us first assume that each rating and thus each edge weight is in $\mathbb{N}$. Link [26] suggests that we model the rating graph as a directed *multigraph*, meaning that more than one edge of the same direction may exist between any two nodes. The integer rating then denotes the number of edges between a rating node and its rated node.

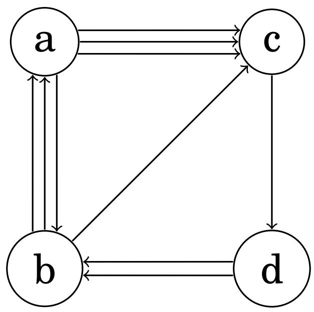


Figure 3.5.: A simple rating multigraph, derived from the graph in figure 3.1

Using the original PageRank equation 3.1, with the only formal modification that $I(p_i)$ no longer denotes a set but a multiset that contains a node as many times as there are incoming edges towards p_i, the PageRank of node c in figure 3.5 is expressed as such:

$$PR(c) = \frac{1-d}{3} + d\left(\frac{PR(a)}{4} + \frac{PR(a)}{4} + \frac{PR(a)}{4} + \frac{PR(b)}{1}\right) = \frac{1-d}{3} + d\left(3 \cdot \frac{PR(a)}{4} + \frac{PR(b)}{1}\right)$$

In general, the PageRank equation for multigraphs can thus be written as

$$PR_{Multi}(p_i) = \frac{1-d}{N} + d \sum_{p_j \in I(p_i)} n_{j,i} \cdot \frac{PR_{Multi}(p_j)}{L(p_j)} \tag{3.2}$$

whereby $n_{j,i}$ denotes the number of edges from node p_j to node p_i, and it becomes clear that a higher rating leads to a proportionally higher relevance flow towards a single node.

3.5.2. Generalization Of Weights Leads Back To Simple Graphs

A simple generalization of the concept to allow non-integer ratings leads us to abandon the multigraph again and instead use the ratings directly as a weight factor to the corresponding incoming node's PageRank. Instead of using the number of outgoing edges of an edge's origin, $L(p_j)$, we must now however use the sum of all of the weights of the outgoing edges of this node. Defining $w_{j,i} \in \mathbb{R}$ as the weight of an edge from p_j to p_i and $O(p_j)$ analogous to $I(p_j)$ as the set of outgoing edges from j, this gives rise to the following formula:

$$PR_W(p_i) = \frac{1-d}{N} + d \sum_{p_j \in I(p_i)} \frac{w_{j,i} \cdot PR_W(p_j)}{\sum_{k \in O(p_j)} w_{j,k}} \tag{3.3}$$

It is easy to see that the multigraph-approach equation in 3.2 is equivalent to 3.3 when constrained to weights within $\mathbb{N}$.

The resulting matrix is real, non-negative and column-stochastic (as the Google-Matrix is).

3.6. The Transition Matrix

PageRank can also be regarded as the expected result of a Markov process [18], in which a transition from one state to another state occurs with fixed probability, independently of past or future transitions. In the original PageRank article this is intuitively explained with the behavior of a *random surfer*. The current state is the current website the surfer is looking at, and the transitions occur by following any of the available links on this website at random. With a probability of $1-d$, d being the aforementioned damping factor, the surfer may also choose to end his link-traversal and choose a completely random website instead, called a *random leap*. The PageRank value of a website is then the probability that the surfer may eventually arrive at that site.

We can fully define a Markov process with a quadratic transition matrix that contains in each entry $t_{i,j}$ the probability that a state transition from j to i occurs. This matrix must be a left stochastic matrix, meaning that in addition to being quadratic, each entry must contain a non-negative number and each column must sum up to 1. Intuitively, this is explained by the realization that the summed up probability of choosing any transition from a given state is 1.

3.6.1. State Transitions In The Original PageRank

Setting aside the random leap factor for now, the transition matrix for the original PageRank is easily obtained by taking the adjacency matrix of the website graph.

To obtain a transition matrix for the Markov process, we transpose the adjacency matrix, and normalize each element by the sum of its column, i.e. the number of outbound links on the node which the column represents, since the random surfer's probability of choosing a link is equal for all links on the current site. (Note that we only obtain a left stochastic matrix if there are no dangling pages, which are pages that contain no links to other pages, as they result in columns consisting of all 0s. However, this technicality will be resolved later on by taking into account random leaps.)

More formally, we can extract the transition matrix using a transformation of equation 3.1, so that we are later able to apply the same approach to the weighted PageRank equation 3.3. For illustration purposes, we first simplify it to a version that only contains the "link traversal" part of the sum, as we will handle random leaping later:

$$PR_{simple}(p_i) = \sum_{p_j \in I(p_i)} \frac{PR_{simple}(p_j)}{L(p_j)} \tag{3.4}$$

Instead of summing only over the nodes with an outgoing edge to p_i, we can also write the equation with a sum over *all* nodes in the graph, by using an indicator function $\mathscr{I}(j,i)$ which is 1 if an edge from j to i exists and 0 otherwise.

$$PR_{simple}(p_i) = \sum_{p_j \in V} \mathscr{I}(j,i) \times \frac{PR_{simple}(p_j)}{L(p_j)} = \sum_{p_j \in V} \frac{\mathscr{I}(j,i)}{L(p_j)} \times PR_{simple}(p_j) \tag{3.5}$$

It is clear that the term $\frac{\mathscr{I}(j,i)}{L(p_j)}$ is a random surfer's probability of clicking a link on site j to site i and thus entry $t_{i,j}$ in the transition matrix.

3.6.2. State Transitions In Weighted PageRank

Following the same approach for weighted PageRank, we simplify its equation 3.3 to only contain the rating part, and rearrange. No new indicator function needs to be defined, as the edge weights act as it if we just assert that the weight $w_{j,i}$ is 0 when there is no edge from j to i. Indeed unweighted PageRank emerges as a special case of weighted PageRank, where the weight of an existing edge is always 1.

$$PR_{W,simple}(p_i) = \sum_{p_j \in V} \frac{w_{j,i} \times PR_{W,simple}(p_j)}{\sum_{k \in O(p_j)} w_{j,k}} = \sum_{p_j \in V} \frac{w_{j,i}}{\sum_{k \in O(p_j)} w_{j,k}} \times PR_{W,simple}(p_j) \tag{3.6}$$

We gather that entry $t_{i,j}$ of the transition matrix is $\frac{w_{j,i}}{\sum_{k \in O(k)} w_{j,k}}$ for weighted PageRank. In other words, it is the weight of the corresponding outgoing edge normalized with the sum of all weights. The matrix is also easily interpreted as a modified adjacency matrix that contains the (normalized) weights instead of just a 0 or 1 which indicates edge existence.

The columns clearly sum up to 1, thus retaining the matrix as a left stochastic matrix and we can follow up with an interpretation of its entries as probabilities within a Markov process.

$$\begin{pmatrix} 0/4 & 2/3 & 0/1 & 0/1 \\ 1/4 & 0/3 & 0/1 & 1/1 \\ 3/4 & 1/3 & 0/1 & 0/1 \\ 0/4 & 0/3 & 1/1 & 0/1 \end{pmatrix} = \begin{pmatrix} 0 & 2/3 & 0 & 0 \\ 1/4 & 0 & 0 & 1 \\ 3/4 & 1/3 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 3.6.: The transition matrix of the rating graph in fig. 3.1

3.7. Student Ratings As Lecturer Choice Probabilities, Lecturer Grades As Student Choice Probabilities

Having asserted that the transition matrix is still a left stochastic matrix, we can now interpret the normalized weights, which essentially represent the relative ratings the participants give out to each other, as probabilities.

There is an intuitive interpretation of ratings from *students* given towards *lecturers*: They can be seen as the probability that a student, after hearing the lecturer's lectures, will prefer attending lectures by that lecturer again, over lectures by other lecturers. Recall that, due to the normalization, the ratings are *relative* to each other, i.e. they are normalized with the sum of all ratings the student has given. This means that in our approach, a single rating, or, from lecturer to student, a single grading, is not absolute and only meaningful if taken into relation with all others of that respective participant. A student that rates a lecturer twice as high as another is stating that, given the choice, he is twice as likely to attend a lecture of the higher rated lecturer again.

The interpretation of the other direction, from lecturer to student, is slightly less intuitive because of the somewhat weaker notion that lecturers try to choose students. However, the interpretation only takes into account the *intent* of the participant, and we just assume that a lecturer would choose higher rated students accordingly if given the possibility.

What further follows from this interpretation is that lecturers, via proxy of the students they have graded, *indirectly* rate *themselves*. Conversely, students indirectly rate each other via their rated lecturers. Incidentally, this is exactly how the particular method we chose to calculate the PageRank values, the Power Method discussed in section 3.9, works.

3.8. The Random Leap Factor

After the in-depth discussion of the incoming nodes' contribution to a node's total PageRank, we will now revisit its other addend, the *random leap factor*.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in I(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{1.1}$$

The random leap factor accounts for a fraction of $1-d$ of the total PageRank, while the remaining fraction of d is contributed by the incoming nodes.

3.8.1. Random Leaps In Original PageRank

In the original PageRank interpretation of the random surfer model, the random leap factor accounts for the fact that a surfer may, at any step during his journey, not choose any available link on the current site, but instead directly visit *any* of the available websites at random, e.g. by typing in its URL. The probability for this behavior is $1-d$, and in the case that this occurs, the probability for choosing a particular page results from a uniform distribution across all available sites, i.e. $\frac{1}{N}$. Thus, under the assumption of a popular damping factor of $d = 0.85$, the surfer will randomly leap with 15% probability, and among 10 available sites, a random leap to a particular site occurs, within any step, with a probability of $\frac{1-d}{N} = \frac{0.15}{10} = 0.015 = 1.5\%$.

In the transition matrix that describes the Markov process of a random surfer, each probability of a state transition during a particular state was simply the probability that a particular link on the current site was taken, which was uniformly distributed among all existing links on that site. Sites that weren't linked from the current site couldn't be reached directly in one step, so their corresponding state transition probability was 0.

With random leaps, since the leap's destination is truly chosen randomly among all existing sites (and thus, states) in the Markov process, it is now possible to reach *every* state from whatever state. Since random leaps and link traversals are mutually exclusive, the overall state transition probability is now just the sum of those two events' probabilities.

$$\begin{pmatrix} \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 2/3 & \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 0 \\ \frac{1-d}{4}+d\cdot 1/4 & \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 1 \\ \frac{1-d}{4}+d\cdot 3/4 & \frac{1-d}{4}+d\cdot 1/3 & \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 0 \\ \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 0 & \frac{1-d}{4}+d\cdot 1 & \frac{1-d}{4}+d\cdot 0 \end{pmatrix} = \begin{pmatrix} 0.0375 & 0.6041.. & 0.0375 & 0.0375 \\ 0.25 & 0.0375 & 0.0375 & 0.8875 \\ 0.675 & 0.3208.. & 0.0375 & 0.0375 \\ 0.0375 & 0.0375 & 0.8875 & 0.0375 \end{pmatrix}$$

Figure 3.7.: The transition matrix in fig. 3.6 modified with the inclusion of random leaps, $d = 0.85$

Note that transition probabilities that were 0 are now $(1-d)\cdot\frac{1}{N}$, and that the matrix's column sums are still 1.

The diagonal of the resulting matrix only contains non-negative non-zero values, ensuring that the Power Method (see below) converges.

3.8.2. Random Leaps In Weighted PageRank

For generalized weighted PageRank, no modification of this concept is necessary, as random leaps are usually regarded separately from weighted or unweighted link/edge traversal. However, weighted PageRank offers a nice alternative interpretation of random leaps that fully incorporates them back into edge traversal: instead of handling random leaps as a type of transition which is separate to edge traversal, they can be construed as implicit edges that exist between every node in either direction. If we correct the weights of every preexisting explicit edge by multiplying it with d, the newly established implicit random leap edges are edges with weight $\frac{1-d}{N}$, where N, being the number of nodes, is also the number of implicit outgoing edges per node. Thus, a random leap now constitutes an ordinary edge traversal, however one that follows implicit edges between every existing node, and with lower probability.

Ding et al. [12] go a step further and propose weighting the nodes itself for random leap, by assigning different random leap probabilities to each node instead of an overall damping factor distributed uniformly across all nodes, and thus giving “more relevant” nodes a higher chance to be reached by random leap. While node weights are also easily construed as differently weighted implicit random leap edges, we are presently not interested in weighting individual nodes (i.e. students and/or lecturers) in any way, so we do not follow this approach.

3.8.3. Random Leaps In eval²Rank

Ranking algorithms in bipartite graphs are generally better examined using the student-lecturer model instead of a random surfer model, as the underlying bipartite graphs consist of two disjoint sets of nodes that actually represent two different sorts of participants, which may not interconnect within themselves.

Taking the interpretation suggested in section 3.7 back into consideration, where the relative ratings of students towards lecturers are taken as probabilities of choosing a lecturer, the idea of leaping randomly is easily incorporated into those probabilities, in a manner akin to the random surfer model. Even if students were given the full choice of lecturers (which of course is hypothetical), it is still hardly conceivable that students would make the decision based on their rating *alone*, which in our model is just a projection of a highly complex, subjective valuation of their lecturers. This fact is similar to the idea that a surfer may suddenly leap to any available website, for reasons unrelated to link traversal.

Random leaps can therefore serve as a means to model the factors of a student’s choice that are not reflected by the one-dimensional rating given into the system. A student may immensely favor one lecturer over every other and, conversely, intend to avoid another one as much as possible. But the mentioned outside factors, lying beyond the scope of what is considerable in a model with viable complexity, may cause him to nevertheless choose a class given by the less favored lecturer.

However, the random leap approach cannot be carried over unmodified. Recall that in section 3.8.2, we established that the concept of random leaping is describable as implicit, lower-weighted edges between nodes. Those edges form a complete graph, meaning that for every node, there exists an outgoing edge to every other node and itself. An implication is that all student nodes are interconnected as well, and so are lecturer nodes. Consequently, the bipartiteness of the extended graph is also not retained.

The problem arises with the fact that connecting the nodes within one partition does not make sense in our interpretation, because it essentially means that students may now choose other students instead of lecturers at random, which is clearly nonsense. We must therefore modify the approach to ensure that choices with probabilities greater than 0 only occur between students and lecturers (and vice versa). In more general terms, just as with regular edges, random leaps may only be permitted between nodes from one partition to nodes from the respective other partition. Under the implicit edge interpretation of section 3.8.2, bipartiteness is then also retained even with the inclusion of random leap edges.

For the bipartite rating graph in fig. 3.3, the bipartite random leap results in the transition matrix shown in fig. 3.8.

$$\begin{pmatrix} 0 & 0 & 0 & \frac{1-d}{3}+d\cdot 3/5 & \frac{1-d}{3}+d\cdot 0/5 \\ 0 & 0 & 0 & \frac{1-d}{3}+d\cdot 1/5 & \frac{1-d}{3}+d\cdot 2/5 \\ 0 & 0 & 0 & \frac{1-d}{3}+d\cdot 1/5 & \frac{1-d}{3}+d\cdot 3/5 \\ \frac{1-d}{2}+d\cdot 3/3 & \frac{1-d}{2}+d\cdot 0/1 & \frac{1-d}{2}+d\cdot 0/2 & 0 & 0 \\ \frac{1-d}{2}+d\cdot 0/3 & \frac{1-d}{2}+d\cdot 1/1 & \frac{1-d}{2}+d\cdot 2/2 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0.56 & 0.05 \\ 0 & 0 & 0 & 0.22 & 0.39 \\ 0 & 0 & 0 & 0.22 & 0.56 \\ 0.925 & 0.075 & 0.075 & 0 & 0 \\ 0.075 & 0.925 & 0.925 & 0 & 0 \end{pmatrix}$$

Figure 3.8.: The full transition matrix including bipartite random leap for fig. 3.3

As before, the column sums are still 1.

3.9. The Power Method

To compute the principal eigenvector of the transition matrix and thus the rankings of our participants itself, we will use the Power Method. According to Haveliwala et al. [17], the Power Method is not only the oldest method of computing the principal eigenvector of a matrix, but also “the heart and motivation of the original PageRank algorithm”.

The Power Method essentially works by just repeatedly calculating $x^{(k)} = \mathbf{A}x^{(k-1)}$, i.e. we start with an arbitrary but non-zero vector $x^{(0)}$ and multiply the transition matrix $\mathbf{A}$ with that vector, repeating the multiplication against the converging vector until the desired precision is reached.

Because matrix multiplication is associative, another way to write down the Power Method's progression is $(\mathbf{A} \times \mathbf{A} \times ... \times \mathbf{A}) \times x^{(0)} = \mathbf{A}^k \times x^{(0)}$. When dealing with bipartite matrices, we can break down the transition matrix into two block matrices, S for ratings given by students and L for gradings given by lecturers. We will examine what happens with those sub-matrices during exponentiation.

$$\begin{array}{c}
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{L} \\
\left(\begin{array}{ccc|cc}
0 & 0 & 0 & 0.56 & 0.05 \\
0 & 0 & 0 & 0.22 & 0.39 \\
0 & 0 & 0 & 0.22 & 0.56 \\
\hline
0.925 & 0.075 & 0.075 & 0 & 0 \\
0.075 & 0.925 & 0.925 & 0 & 0
\end{array}\right) \\
\mathrm{S} \qquad\qquad\qquad
\end{array}$$

Figure 3.9.: Breaking down the transition matrix given in 3.8 into sub-matrices

The consecutive exponentiation of $\mathbf{A}$, written with block matrices and where $\mathbf{0}$ is the appropriately sized all 0 element matrix, is:

$$\begin{aligned}
\mathbf{A} &= \begin{pmatrix} \mathbf{0} & \mathbf{L} \\ \mathbf{S} & \mathbf{0} \end{pmatrix}, \\
\mathbf{A}^2 &= \begin{pmatrix} \mathbf{LS} & \mathbf{0} \\ \mathbf{0} & \mathbf{SL} \end{pmatrix}, \\
\mathbf{A}^3 &= \begin{pmatrix} \mathbf{0} & \mathbf{LSL} \\ \mathbf{SLS} & \mathbf{0} \end{pmatrix}, \\
\mathbf{A}^4 &= \begin{pmatrix} \mathbf{LSLS} & \mathbf{0} \\ \mathbf{0} & \mathbf{SLSL} \end{pmatrix}, \\
\mathbf{A}^5 &= \begin{pmatrix} \mathbf{0} & \mathbf{LSLSL} \\ \mathbf{SLSLS} & \mathbf{0} \end{pmatrix}, \\
\mathbf{A}^6 &= \begin{pmatrix} \mathbf{LSLSLS} & \mathbf{0} \\ \mathbf{0} & \mathbf{SLSLSL} \end{pmatrix}, \\
&\dots \\
\mathbf{A}^{2n} &= \begin{pmatrix} (\mathbf{LS})^n & \mathbf{0} \\ \mathbf{0} & (\mathbf{SL})^n \end{pmatrix}, \\
\mathbf{A}^{2n+1} &= \begin{pmatrix} \mathbf{0} & (\mathbf{LS})^n\mathbf{L} \\ (\mathbf{SL})^n\mathbf{S} & \mathbf{0} \end{pmatrix}
\end{aligned}$$

Note that for even exponents $2n$, the upper left and lower right quadrant are just $(\mathbf{LS})^n$ and $(\mathbf{SL})^n$ respectively. Now let $\mathbf{A}^*$ be the matrix resulting from exponentiation of the transition matrix $\mathbf{A}$ an arbitrary but even number of times, then we get an arbitrarily close approximation of the principal Eigenvector, and thus the ranking, by multiplying with any non-zero vector $x^{(0)}$. We choose a vector of all ones and arrive at:

$$\mathbf{A}^* \times x^{(0)} = \begin{pmatrix} (\mathbf{LS})^* & \mathbf{0} \\ \mathbf{0} & (\mathbf{SL})^* \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \vdots \end{pmatrix} = \begin{pmatrix} r_a \\ r_b \\ r_c \\ \vdots \\ r_A \\ r_B \\ \vdots \end{pmatrix}$$

Whereby $r_a, r_b, r_c, \ldots, r_A, r_B, \ldots$ are the (approximated) participant's rankings (recall that in our example, we used lower case letters $a, b, c\ldots$ for students and upper case letters $A, B\ldots$ for lecturers).

However, suppose we are only interested in a ranking of lecturers, not of students. Because the matrices $(\mathbf{SL})^*$ and $(\mathbf{LS})^*$ are quadratic matrices with a row length corresponding to the number of students (for $(\mathbf{LS})^*$) or the number of lecturers (for $(\mathbf{SL})^*$) respectively, we can just omit the multiplication with $(\mathbf{LS})^*$:

$$\begin{pmatrix} (\mathbf{LS})^* & \mathbf{0} \\ \\ \mathbf{0} & (\mathbf{SL})^* \end{pmatrix} \times \begin{pmatrix} \mathbf{x}_\mathbf{S}^{(0)} \\ \\ \mathbf{x}_\mathbf{L}^{(0)} \end{pmatrix} = \begin{pmatrix} (\mathbf{LS})^* \times \mathbf{x}_\mathbf{S}^{(0)} + \mathbf{0} \times \mathbf{x}_\mathbf{L}^{(0)} \\ \\ \mathbf{0} \times \mathbf{x}_\mathbf{S}^{(0)} + (\mathbf{SL})^* \times \mathbf{x}_\mathbf{L}^{(0)} \end{pmatrix} = \begin{pmatrix} (\mathbf{LS})^* \times \mathbf{x}_\mathbf{S}^{(0)} \\ \\ (\mathbf{SL})^* \times \mathbf{x}_\mathbf{L}^{(0)} \end{pmatrix} = \begin{pmatrix} \mathbf{r}_\mathbf{S} \\ \\ \mathbf{r}_\mathbf{L} \end{pmatrix}$$

$$(\mathbf{SL})^* \times \mathbf{x}_\mathbf{L}^{(0)} = (\mathbf{SL})^* \times \begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} = \begin{pmatrix} r_A \\ r_B \\ \vdots \end{pmatrix}$$

Thus, the actual algorithm to compute the lecturer part of the Eigenvector of the transition matrix as discussed in the preceding chapter, consists of the following steps:

1. Extract the sub-matrices **S** and **L** from the transition matrix,
2. calculate the matrix **SL** by performing a matrix multiplication of the two sub-matrices,
3. repeatedly multiply **SL** against a vector whose length corresponds to the number of lecturers and whose rows are initially all set to 1, until the desired precision has been achieved.

Similarly, the ranking of students can be computed by just using the matrix **LS** correspondingly.

Unlike the original full matrix $\left(\begin{smallmatrix} \mathbf{0} & \mathbf{L} \\ \mathbf{S} & \mathbf{0} \end{smallmatrix}\right)$, neither **S** nor **L** contain any zeros. Since all cells of **S** and **L** strictly positive, **SL** and **LS** also do not contain any zeros.

4. Applying eval²Rank To Student-Lecturer-Ranking

We have so far construed the general approach of eval²Rank. However, in order to actually work with Student-Lecturer-Ratings, some details are missing that we will now fill out.

4.1. Students May Only Rate Visited Lectures

One important base requirement of our platform is that students may only rate lectures that they have attended. Otherwise, students that have not attended a past lecture could still rate it in order to skew the results.

We implement this requirement by only allowing students to give ratings to lectures for which they have been assigned a grade in their own transcript of records, ensuring that they have indeed participated in an exam. This poses an additional restriction on the graph, which we will use later.

Because this restriction does not contradict the previous graph definition in any way but merely extends it, we formulate the restriction as an additional property to the graph:

4. Students may only rate lecturers who have rated them, i.e. an edge $e = (s \in S, l \in L)$ may only exist in E if E also contains an edge $e' = (l \in L, s \in S)$. This property is, however, a technicality in our particular situation, and the algorithm's properties still hold if it is omitted.

4.2. Lecturers Are Rated Through Lectures

Note that in the actual system, students do not rate *lecturers* as a whole but individual *lectures*. Consequently, an even stronger property than property 4 holds: students may only rate those lectures in which they have received a grade, i.e. taken part in some kind of examination. Within this work, however, our algorithm does not distinguish individual lectures but only operates on averaged ratings from one student towards one lecturer and vice versa. This averaging is done by simply taking the arithmetic middle of all ratings between two participants, once for each direction. Consequently, the lectures themselves are not part of the graph, and instead an edge from student to lecturer exists once the student has rated at least one of the lecturer's courses.

Instead of averaging the lectures out, one might consider another algorithm suited for *tri*partite ranking (which might consequently be called *eval³Rank*), in which the lectures still remain nodes of the rating graph. Refer to section 9.1 for details.

4.3. Dangling Nodes

As with PageRank, we need to prevent any dangling nodes, which are nodes that do not have any edge to any other node, as they create columns with a sum < 1 in the transition matrix (discussed in 3.6).

Brin and Page [7] propose the removal of dangling nodes prior to the actual PageRank calculation. Eiron et al. [14] note that this requires a recursive algorithm in order to remove any nodes which become dangling after each removal step. Besides that, removing dangling websites prior to ranking is not suitable for today's reality of the Web and they therefore suggest other methods of incorporating them.

In our student-lecturer situation however, removing dangling nodes is indeed viable and just consists of the following two steps:

1. We first remove any *unrated* lecturers, i.e. lecturers which have received no rating at all, as it is not meaningful to rank them among others in any way. Removing them does not create any new dangling students: if there would have been an incoming rating from a student, this step would not remove the lecturer node.

2. Dangling students, in the form of students that have received grades but not rated anybody, are also safely removed. No new dangling lecturers are created: Recall that as of property 4, a rating implies a grading, so either the lecturer has outgoing grading edges as well as incoming rating edges, or it was already removed as unrated in the previous step.

Unrated *students* are of no further concern. It follows from property 4 that any student that has given any rating is also graded. Thus, a student that is unrated must not have given out any rating. The student is therefore also a *dangling* student and removed in step 2.

4.4. Handling Zeros

Looking at the transition matrix of a Student-Lecturer Rating Graph, in the state before any transformations such as adding random leaps or normalization, a 0 may appear in a matrix cell for the following reasons:

1. The cell denotes a potential edge between two members of one partition, i.e. between two students or between two lecturers. Since the graph is strictly bipartite, no such edge exists, and all cells denoting edges between nodes belonging to the same partition are 0.

2. The column or row represents an unrated or dangling participant, and thus all cells within its column or row are 0.

3. The cell represents a rating between two participants from differing partitions, but the respective lecturer has not graded the student or the respective student has not graded the lecturer.

Zeros that occur because of reason 1 are fully intended because a direct rating between members of one partition does not make sense in our context. Zeros that occur because of reason 2 are unproblematic, because they are removed early according to the last section. We will thus look how to handle zeros that occur because of reason 3, i.e. zeros that are results of participants that aren't completely unrated or dangling, but unrated by particular participants of the other partition. In practice, we expect this to be a common occurrence, since almost no student will partake in and rate a lecture of all existing lecturers.

Recalling section 3.8, such zeros will not remain zeros, because the random leap introduced into the graph will assign at least the small weight resulting from the random leap probability. However, taking that small weight in context to other, bigger weights that directly result from an actual rating, this small weight seems highly unfair: because the weights represent the rating from one participant to another, *relative* to all given ratings of the rating participant, the small weight translates to the misrepresentation that intentionally *unrated* participants are effectively rated *much less* than any other rated participant, especially even less than participants that have been intentionally rated with the lowest value!

In order to remedy this misrepresentation, we will instead replace all zeros that occur because of a missing rating by the average of all the ratings that the respective other participant has given out. Consequently, unrated participants are rated neither particularly good nor particularly bad by the rating participant, but lie exactly in his or her middle.

This step occurs after the removal of dangling and (completely) unrated nodes, but before incorporation of the random leaps.

4.5. Representation Of Ratings And Grades

From an outside perspective, the system needs to work with at least two different scales:

- Students that rate lectures use a "5-star-input", where they essentially just pick an integer rating between 1 and 5 (1 being worst, 5 being best).
- Students are in turn graded by their university, which uses one of a multitude of existing grade systems throughout the world. We currently use the grade system common to German universities, as the only available backend is an institute-specific transcript upload.[1]

However, for the algorithm to work, we need to transform all ratings and gradings to one common scale. Instead of choosing one of the existing scales as the internally used rating scale, we established a new, easily-represented 10-point scale. We do this for the following reasons:

- Decoupling the scales from the algorithm. The goal of the system is to support all different kinds of rating and grading scales. By using a separate "neutral" scale, we avoid confusion that may occur while transforming the scales into each other. Instead, all values are projected onto the 10-point-scale as the sole representation for all kinds of ratings and gradings.
- Linearization. We would like our internal scale to be as "linear" to a participant's performance (whether it be studying or lecturing) as possible. However, at least the grade

[1] The German university grade system works as follows: a grade is one of the discrete values 1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.7, 4.0 and 5.0, whereas 1.0 is the best and 5.0 the worst grade. Grades with .7 as decimal place are the "plus" variant of the next rounded up integer, e.g. 2.7 can be referred to as 3+. Similarly, grades with .3 as decimal place are the "minus" variant of the integer when rounding down, e.g. 2.3 can be referred to as 2-. For 1.0 only the "minus" variant 1.3 exists, for 4.0 only the "plus" variant 3.7 exists. For 5.0, no variant exists, as it is the grade generally assigned in case of failure (consequently, 4.0 is the last grade that passes).

Rating Stars	Rating Points
1	2
2	4
3	6
4	8
5	10

Table 4.1.: Mapping between rating stars and rating points

system for German universities is not a linear projection of a student's performance in an examination. As mentioned in the next section, it is also not entirely clear how a student's ratings is best placed on such a scale.

This 10-point scale is simply represented by floating point numbers between 0 and 10 inclusive, whereby 10 is the best rating. Note, however, that 0 is not the worst available rating, as, according to section 4.4, zero means *unrated*. The worst available rating would be the lowest number above zero that the chosen floating point data type is able to represent. This is only a technicality, however, as the original scales we are transforming from are usually discrete in some way or another, and thus resulting in some other fixed value above zero.

The 10-point scale is only an internal representation, and the points may be mapped to any other floating point range if the situation calls for it. As seen, in the Web application's user interface, both grades and ratings are mapped to a visually more practical representation of 5 stars.

4.5.1. Mapping Rating Stars To Rating Points

For the ratings, we just map the amount of stars given (out of 5) linearly to the points, as seen in table 4.1. Note again that the lowest rating of 1 star does not correspond to 0 points but that the 1 star corresponds to 2.0 points on the internal scale.

It is unclear, however, if such a linear mapping is really satisfactory. Since we would like the 10-point-scale to reflect a linear projection of students' opinions, we hereby also assume that a student gives the "middle" rating, i.e. 3 stars[2], to lecturers whose performance he or she really deemed "average". However, this may not be the case, and in future work, an appropriate mapping may be construed by considering work from other research areas (e.g. psychology).

4.5.2. Mapping Grades To Rating Points

For the grades, the mapping is not as straightforward. While a linear mapping from the existing grade scale between 1.0 (best) and 5.0 (worst) is thinkable, it does not reflect a linear measurement of the student by the very definition of the German grade system. A student that received grade 5.0 (failed) may be much farther away from a student given 4.0, the last passing grade, than a student with 4.0 is to a student with 3.0. This non-linear distance results from the fact that a 5.0 is universally given to any student that failed the respective exam in any way, while the last passing grade 4.0 is supposed to be somewhere around the

[2] assuming, due to normalization, that the student also uses the whole rating spectrum from 1 to 5 stars for other ratings

Grade	Alt. Representation	Rating Points
1.0	1	10
1.3	1-	9.5
2.7	2+	8.834
2.0	2	8.334
2.3	2-	7.834
3.7	3+	7.167
3.0	3	6.667
3.3	3-	6.167
3.7	4+	5.5
4.0	4	5.0
5.0	5	2.5

Table 4.2.: Mapping between grades (German) and rating points

middle of the original rating scale of that exam (often points distributed across the single problems within the exam).

Thus, in average over all exams, a 4.0 is awarded to students who achieved an amount of exam points around 50% or slightly more, while any student with a 5.0 may lie anywhere between 0% and 50% of the exam point scale.

We remedy the situation by mapping in a way that distributes the passing grades 1.0–4.0 to the upper half of the 10-point rating scale and places the failing grade 5.0 in the middle of its lower half. Note that the actual formula is slightly more generalized, in that it allows, in theory, any real number between 1.0 and 6.0 and maps them accordingly to one of both halves. In practice, however, the German grade system is limited in the discrete set of available grades, and table 4.2 shows how those individual grades are mapped. The formula to calculate any linear rating scale from a grade is:

$$p(g) = \begin{cases} p_{max} - \frac{g-1.0}{3.0} \cdot 1/2 \cdot p_{max} & \text{if } 1.0 \leq g \leq 4.0 \\ 1/2 \cdot p_{max} - \frac{g-4.0}{2.0} \cdot 1/2 \cdot p_{max} & \text{if } 4.0 < g \leq 6.0 \end{cases} \tag{4.1}$$

Other grade systems should be mapped accordingly, and it is the transcript backend's responsibility to do so.

5. The Web Rating Platform `eval`

This chapter describes the Web Application that was designed and implemented as part of this thesis. The Web Application embodies a fully functional platform which allows students to enter grading and rating data and in turn creates and presents rankings based on the input data, using the algorithm detailed in the last chapters. The current chapter focuses on goals and usage of the Web Application. Details about the implementation of the Web Application, as well as differences and extensions of the actually deployed prototype, are subject of the two chapters following this one.

At the time of this thesis' completion, the prototype is readily deployed at `https://eval.pms.ifi.lmu.de`.

5.1. Name

This project is about evaluation in multiple senses: Traditionally, universities evaluate students, and in this project, these evaluations represent the rating graph's edges from lecturer to student. The primary goal of this project, on the other hand, is to materialize the graph's complementary edges, which denote ratings from student to lecturer, and allow the students to feed back evaluation themselves into the system. Last but not least, the algorithm that creates ranking from this graph is subject to evaluation as well, for which the Web Application serves as a means to gather data and test the proposed or future algorithms.

Because of that, the name `eval` has been selected. The monospaced serif typeface has been deliberately chosen as a "shout out" to an often seen convention of formatting source code tokens within text paragraphs, a convention that is also used by this thesis itself. As such, `eval` represents a symbol often found in interpreted computer languages, usually as the identifier of a function that evaluates its arguments as code in the language itself, and should invoke familiarity in the primary target audience of the Web Application's first iteration–the department of computer science at LMU. While the term "evaluation" is hereby usually meant in the sense of evaluating an expression or statement within the interpreter, we borrow the symbol for the intended meanings described in the last paragraph.

5.2. Requirements

To provide a fully usable evaluation platform and enable this platform to make use of the described algorithm, `eval` should fulfill the following requirements:

- Students should be able to make their performances, i.e. their grades, known to the application.
- Students should be able to rate lectures. For reasons of fairness and to avoid a deliberate skewing of the ratings, students should only be able to rate lectures that they officially

attended.

- The Web Application should generate rankings, and make these rankings available to everyone interested in them.
- At any point, it is important that students can revoke any information they have fed into the system. This includes ratings as well as grades.
- However, in order to allow the observation of changes over time, snapshotted, *aggregated* data should always be available.

We will briefly explain how each of those requirements are fulfilled by `eval`, and give a full tour of the user interface afterward.

5.2.1. Importing Grades

The first requirement, making the students' grades known to the application, is achieved by letting students import their *transcripts* into `eval`. In the context of this thesis, transcripts are defined as sets of grades assigned to visited lectures. To prevent deliberate cheating of the ranking algorithm by using bogus grades, the transcripts should come for an authoritative source, most likely the API of a university backend. Note that the prototype does not yet communicate directly with a university backend but relies on user-uploaded transcript XML files. This of course violates the authoritativeness requirement and hence the uploaded data should not be ultimately trusted, but still allows us to fully test the Web Application and evaluate the algorithm's output based on its given input.

Students must always approve the import of any transcript (in the prototype, this is implicitly done by manually uploading the transcript). Import is done through an extra page for managing transcripts, which also allows browsing them with all their associated attendances, and revoking them.

5.2.2. Rating Lectures

Students may rate lectures by browsing them on a page which lists all lectures and having the possibility to give each attended lecture a rating. To allow students to find lectures that are relevant to them, they can set various filters, e.g. for filtering according to the lecturer's or lecture's name, semester, and whether they have already rated the lecture or not. The list of lectures is also paginated (currently 200 lectures are displayed at a time) to avoid long load times over slow connections, though the page size can be chosen arbitrarily in the code.

Ratings are presently given on a simple one-dimensional scale: similar to the user interface for rating implemented in e.g. media players, students choose their rating between 1 and 5 by clicking on the corresponding star (see fig. 5.1).

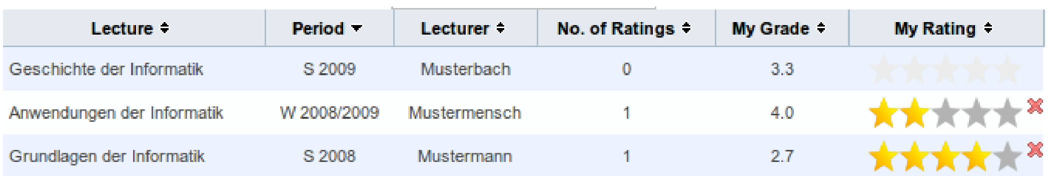


Figure 5.1.: The user interface used to rate lectures

As mentioned, only attended lectures should be rateable for a student, and therefore the rating column is only populated for lectures for which an attendance has been reported through an imported transcript.

Note that the list of available lectures and associated lecturers in the database are completely build up from the imported transcripts, no separate data entry is necessary.

5.2.3. Rankings

Rankings are generated for students *and* lecturers separately. However, only lecturer rankings are publicly displayed. For student rankings, only the student's *own* ranking, as well as the total number of ranked students for reference, are available to each respective student.

The implementation of the rating algorithm as described in the preceding chapters always operates on the current state of the database upon invocation. A new ranking is generated every morning at 6:00am without any user interaction.

Rankings are made public and thus available to *all* users, whether logged in or not. To prevent issues with session management and user state, permanent URLs are generated for each ranking.

5.2.4. Revoking Information

Students can revoke any rating any time by hitting the red x-mark seen in figure 5.1. The change is effective immediately. Students are also free to re-rate lectures with revoked ratings anytime.

The same applies for transcripts: in the transcript manager, each imported transcript features a "Revoke" button. This button revokes the transcript and all its associated attendances immediately from the system. If the associated attendances included any rated lectures, the ratings are also implicitly revoked. To prevent accidental revocation of transcripts and all associated ratings, a confirmation dialog box appears prior to performing the operation.

5.2.5. Preservation Of Aggregated Snapshots

Aggregated Snapshots are made available in the form of the rankings itself, by simply preserving each generated ranking. The ranking page not only shows the most recent ranking, but allows any user to select a previous ranking for display. See section 5.2.3 for details.

5.3. The User Interface

`eval`'s user interface is divided into several sub-pages which group together any content related to the different aspects of the Web Application, e.g. transcript management, lecture rating and rankings.

Some of these sub-pages are only accessible to logged in students, while others (specifically the Login and Register sub-pages) are only accessible to users who have not yet logged in. At any time, all available sub-pages are available by a menu, which is always visible on the left side of the page's content.

In this section, we will show a screenshot of each sub-page (populated with example data) and describe its function. For detailed explanation of the associated views with a focus on implementation, see section 6.4.

5.3.1. Home

When not logged in, the Home sub-page, which is also the site's index page reached when visiting `eval`'s root URL, simply contains a short welcome message and links to the Rate page ("browse") and the Login page.

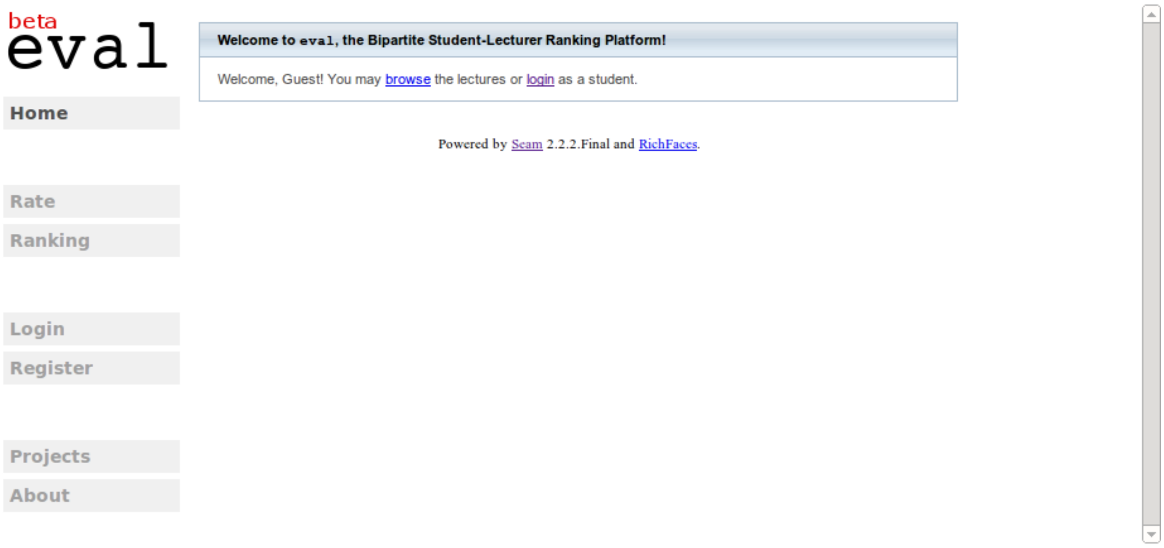


Figure 5.2.: The Home Sub-page, when not logged in

When logged in, a longer welcome message that describes how to use `eval` is displayed instead. If the student has been ranked, his or her rank is displayed as well (remember that *only* the student's own rank is ever displayed, and only to the respective student).

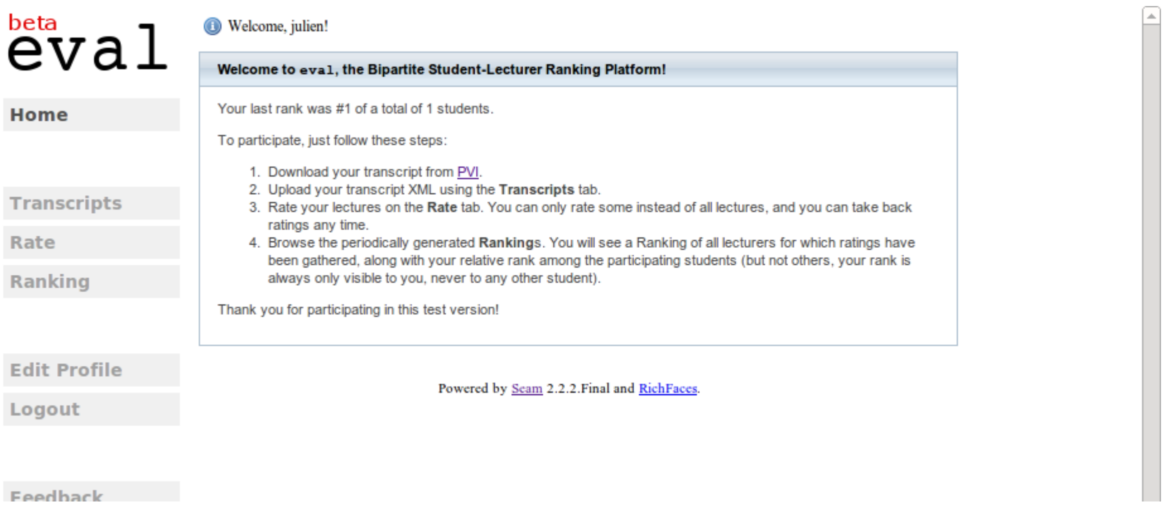


Figure 5.3.: The Home Sub-page, when logged in

5.3.2. Login

The Login sub-page, which is also reached indirectly when trying to perform restricted actions without being logged in, simply prompts the user for his username and password. In the prototype, the page also contains a direct link to the Register sub-page.

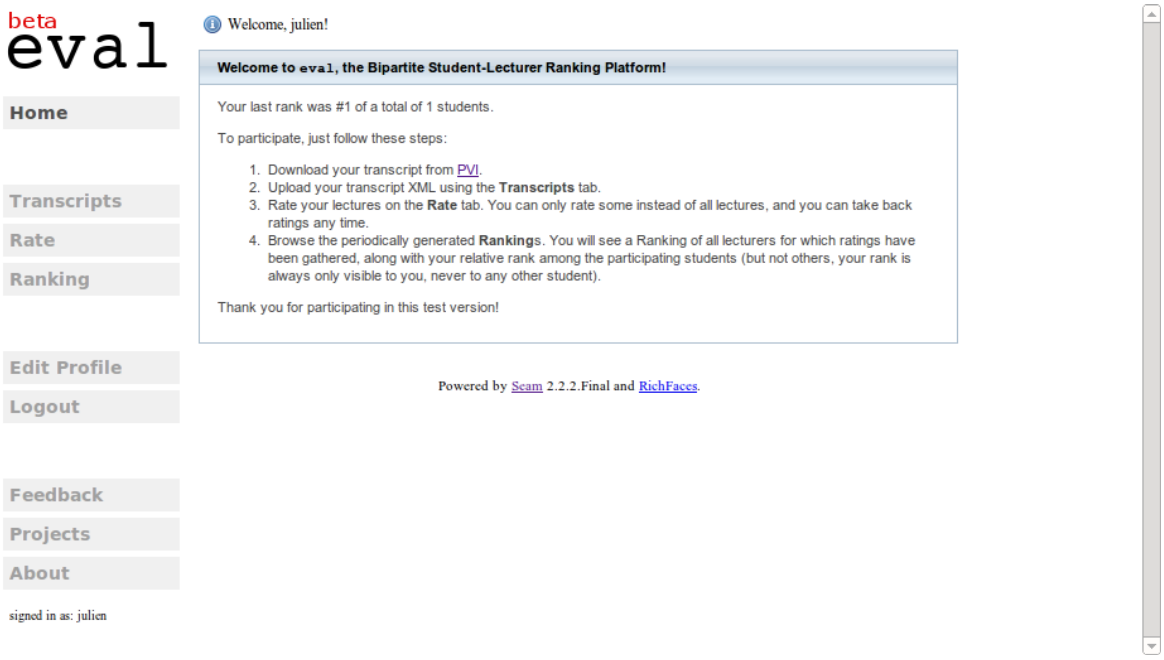


Figure 5.4.: The Login Sub-page

5.3.3. Transcripts

This page is only available to logged in students. It simply presents a list of transcripts imported by the student. Each transcript is displayed with its date of import and a "Revoke" button, that allows students to revoke the transcript, all its attendance records and all associated ratings after a quick confirmation.

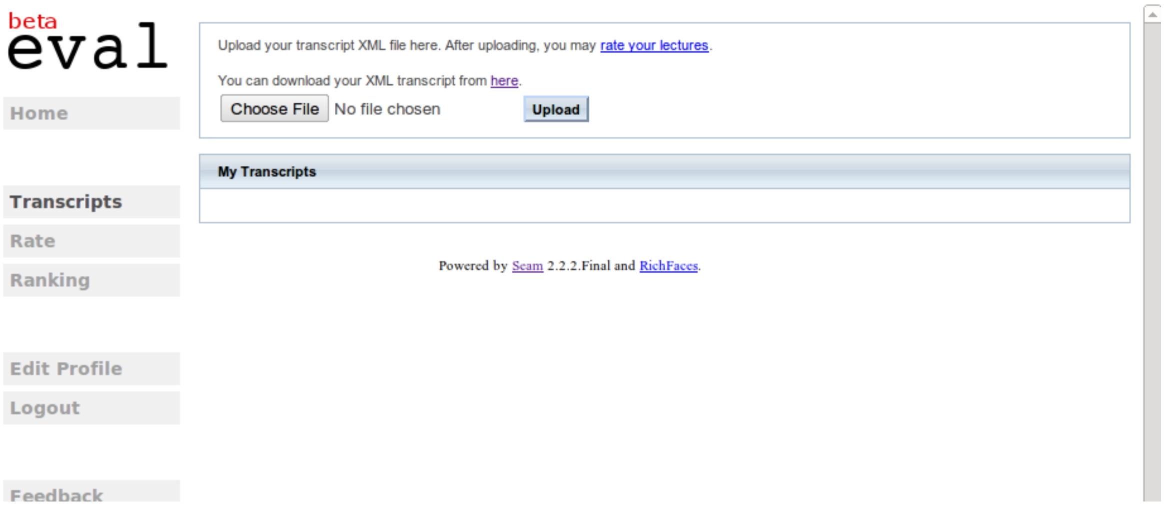


Figure 5.5.: The Transcripts Sub-page with no imported transcripts

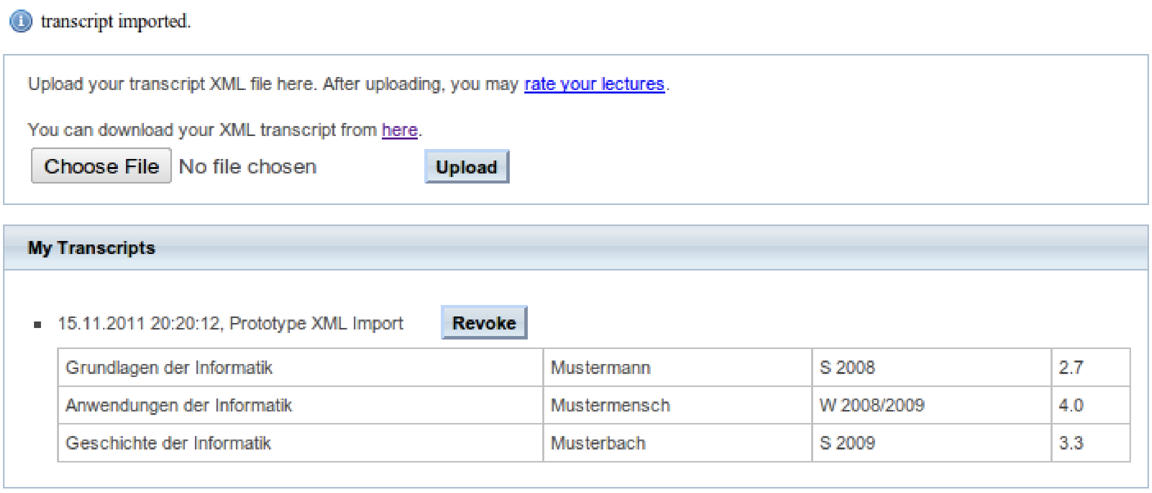


Figure 5.6.: Excerpt of the Transcripts page, after transcript import

In the prototype, as seen in figures 5.5 and 5.6, transcript upload controls are also present. See chapter 7 for details.

5.3.4. Rate

The Rate sub-page allows students to browse lectures and rate them. The central element of the page is the lecture list, which displays all or only filtered lectures, sorted by the order specified by clicking on the table headings. By default, lectures are sorted by semester in descending order, to show the most recent lectures first.

Lectures may be filtered by typing in parts of the lectures' and/or the lecturers' name and pressing the associated Set button, by selecting a semester from the semester drop-down (taking immediate effect), and by choosing one of the "All", "Unrated" or "Rated" links on the top right corner of the page.

If there are more lectures to display than the maximum page size allows (currently set to 200 lectures), a data scroller above the table allows users to navigate pages.

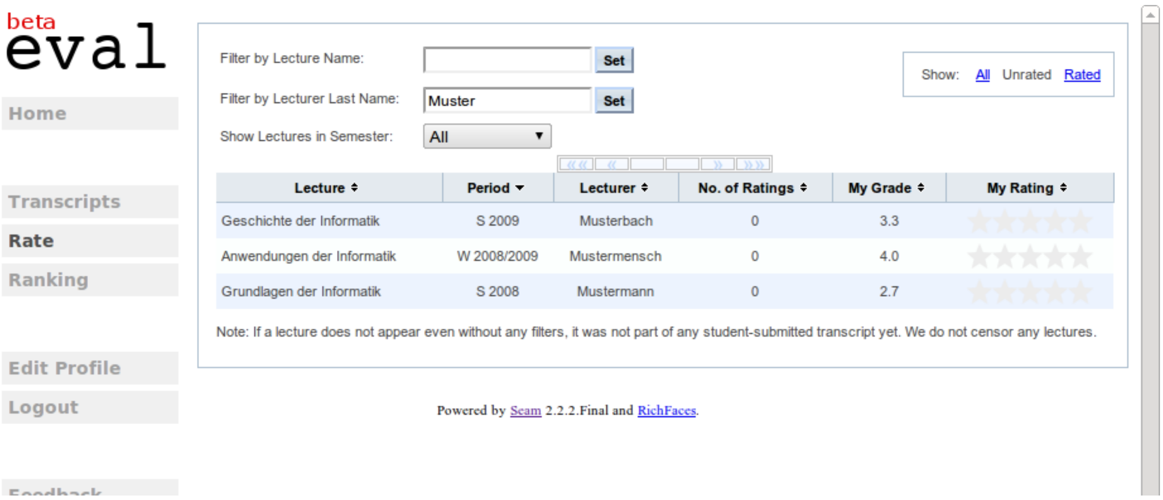
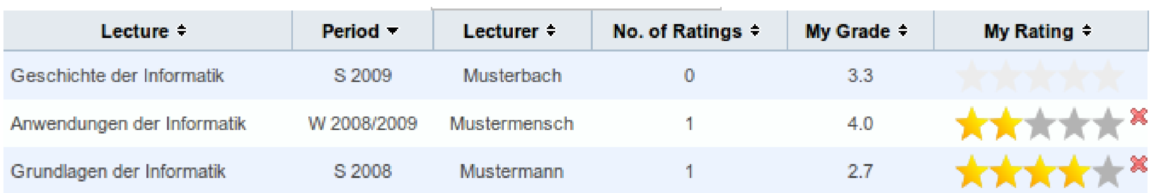


Figure 5.7.: The Rate page

For logged in students, the "My Rating" column on the far right is visible, and attended lectures may be rated by simply clicking on the appropriate star. Rated lectures are immediately displayed with the amount of filled out stars that the student has chosen, and an additional red x-mark that allows revocation of the rating.



Lecture	Period	Lecturer	No. of Ratings	My Grade	My Rating
Geschichte der Informatik	S 2009	Musterbach	0	3.3	
Anwendungen der Informatik	W 2008/2009	Mustermensch	1	4.0	
Grundlagen der Informatik	S 2008	Mustermann	1	2.7	

Figure 5.8.: Rated lectures

5.3.5. Ranking

The Ranking page, available to everyone, shows the currently selected (or most current if none has been selected yet) lecturer ranking. The ranking is displayed by a simple table that contains the lecturer's rank, name, and the calculated eigenvector value.

A permanent link below the ranking table allows sharing of the currently displayed ranking.

Below the panel that contains the rating table, a list of all available lecturer rankings is shown, together with their creation time. These links can be clicked to change the currently displayed lecturer ranking.

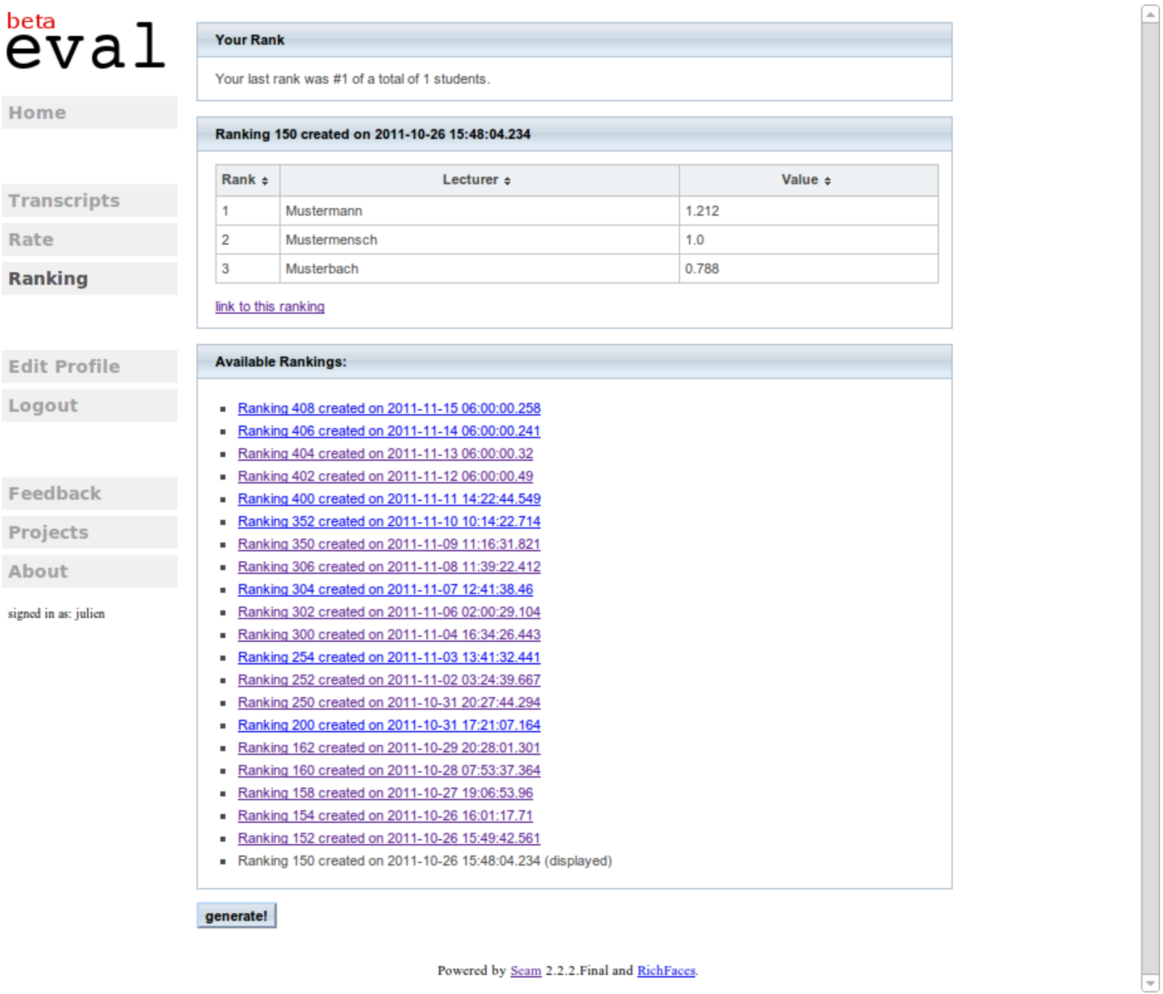


Figure 5.9.: The Rankings Page

Computation currently normalizes rankings using normalization with a maximum-norm on the eigenvector[1], resulting in the highest lectured shown with a value of 1.0, and all other lecturers in proportion to it.

The "generate!" button on the page's bottom is only available to administrators and requests immediate generation of new student and lecturer rankings. Otherwise, rankings are created automatically each morning at 6:00am.

[1]not shown in the picture, because its examplary ranking was generated with a different approach.

5.3.6. Edit Profile

This is a page that mostly exists for future extensions, in which additional profile data should be associated with each user. It currently contains simple radio-boxes with which users may choose their gender, if they prefer to do so. While the gender is currently unused, it is persisted with the user profile residing in the database and may be used for external analysis.

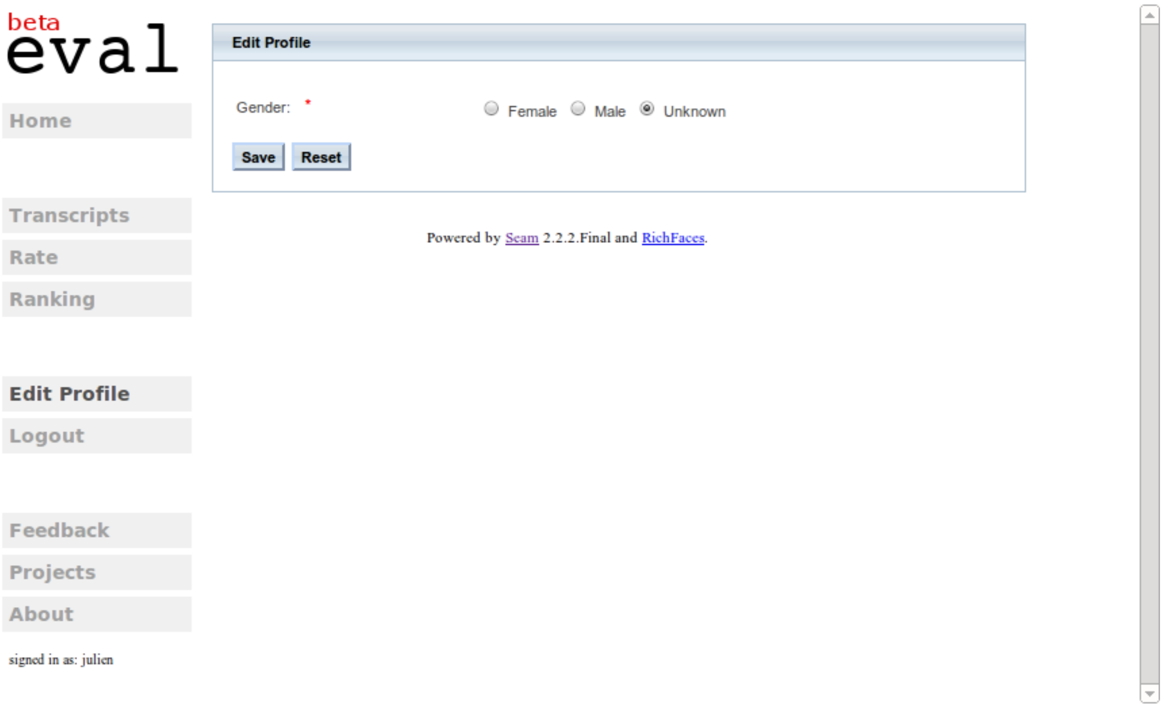


Figure 5.10.: The Edit Profile Page

Additionally, a *nickname* exists for each user, reserved for social features like commenting. However, since the nickname is currently unused as well, the form element is presently not displayed and hardwired to the user's username.

The Edit Profile Page is only available to logged in users.

5.3.7. Feedback

The Feedback Page allows logged in users to submit entered feedback. The feedback is anonymously sent to a hardwired email address, set to `eval@pms.ifi.lmu.de`. However, to prevent spamming, the user must be logged in.

Figure 5.11.: The Feedback Page

5.3.8. Projects and About

Projects and About are completely static pages available to everyone. They contain information about the system, such as associated people and associated projects and the Teaching and Research Unit that `eval` originated from.

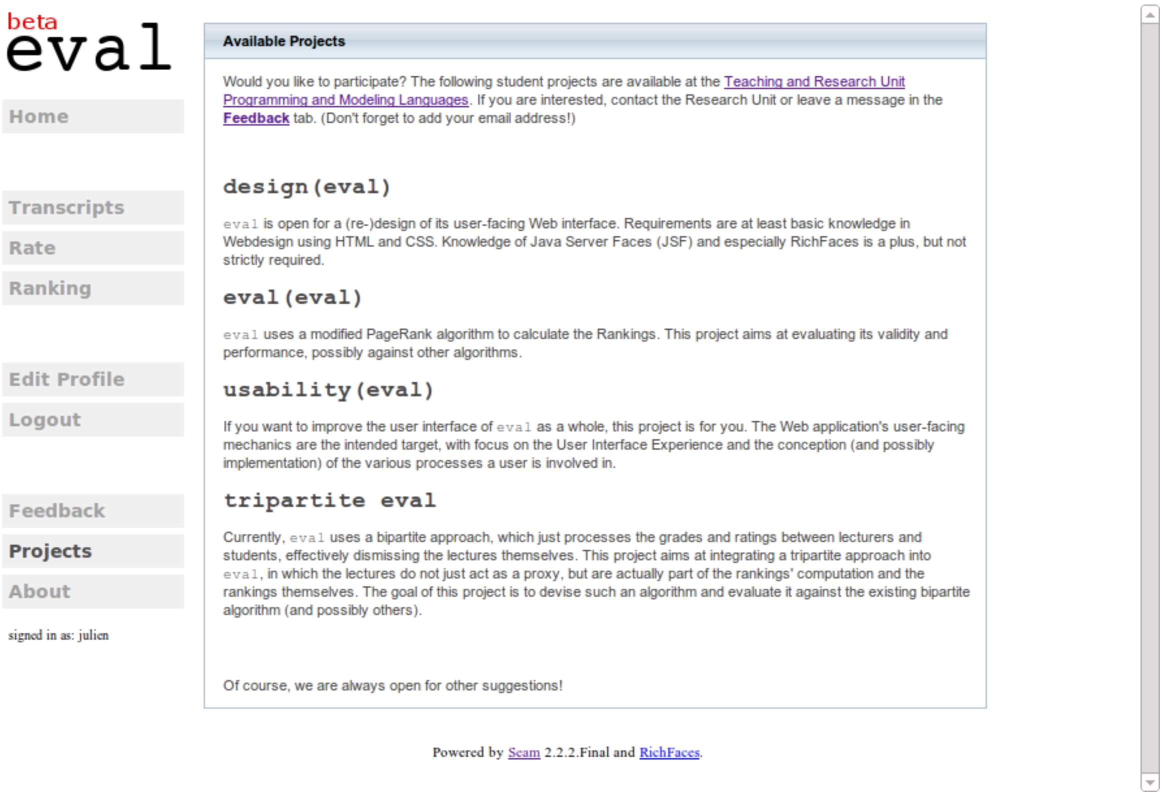


Figure 5.12.: The Projects Page

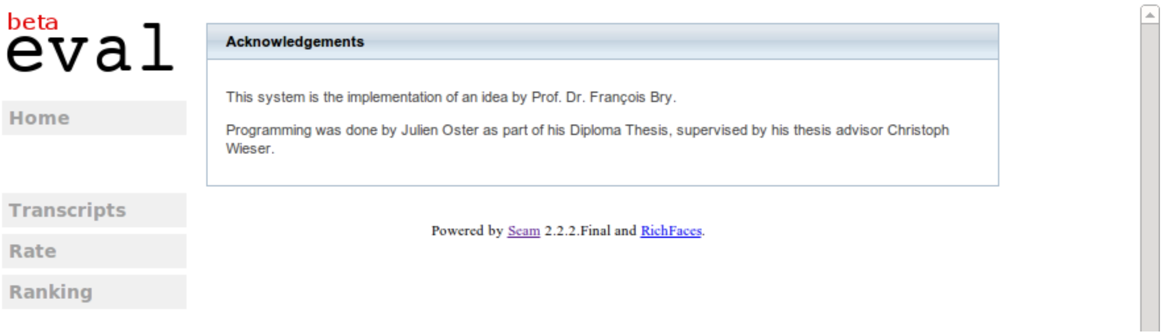


Figure 5.13.: The About Page

6. Implementation of `eval`

For the implementation of `eval`, we chose the *JBoss Seam* Web Application framework, because it allows for flexible and robust, yet easy development of rich, dynamic Web Applications [33]. Additionally, it has been successfully used in past and present projects at the Teaching- and Research Unit that this thesis originates from, meaning that the required know-how to continue the development of `eval` is already available.

Seam presents itself as a frame work on top of Java EE, presenting an MVC (Model-View-Controller) approach to interactive user interface application design, as first described by Krasner and Pope [24]. Consequently, we will describe `eval`'s implementation divided into sections for Model, View and Controller, although we prefer the order Model-Controller-View, as we feel that this allows a more natural exploration of the implementation from "bottom" (the data model, section 6.1) to "top" (the user interface views, section 6.4), with the business logic as Seam Components (section 6.2 and 6.3) in between.

6.1. The Data Model

The foundational bipartite, directed graph is implemented as entities in a relational data model. Additionally, a surrounding support structure of entities implements the data store for a fully fledged Web application, representing the hierarchy behind Universities, Students and Lectures.

The data model is designed to be as simple, redundancy-free and yet extensible as possible. With one exception (described later) it is at least in 4th normal form [23]. The result is a data model that is not only expressive and obvious to work with, but also one that maps naturally to simple Java classes suitable for Object-Relational Mapping, as described in the next section.

The current version of the application is targeted at one single institute of one university. This also means that there is just one backend for student identification and authentication, as well as only one backend for transcript import. However, in order to allow a potential expansion to other institutes or even other universities, we created small "skeleton entities" for entities with a potentially higher multiplicity than one, specifically the university, the transcript source and the authentication backend. These skeleton entities currently hold no data besides their primary key identification and possibly a human readable description. In current use, there is also only one actual instance per entity. However, when the need arises, further fields can be added to the entities and further instances can be created, so as to broaden the target user base of the application without the need for a costly refactoring and any resulting data conversion.

6.1.1. Entities as Simple Java Classes

All data, with the exception of (at most) session-scoped Web application state, are persisted in a relational database. However, the application code does not directly query or modify the database by issuing SQL statements and, in case of a query, iterating over the given result-set. Instead, we use JPA, the Java Persistence API (and specifically Hibernate as an implementation thereof) to bypass any hand-written marshalling code and transparently retrieve, create and modify plain Java objects. It is then the persistence layer's responsibility to manifest the resulting SQL schema by mapping those classes and their relationships to tables of the relational database.

Consequently, a developer may freely modify the data model's structure by only modifying the Java classes used to handle the actual data. There is no need to manually create or maintain the schema on the SQL RDBMS, as it is procedurally generated or updated by the framework. This approach speeds up the development cycle and prevents inconsistencies between code and database schema.

We therefore designed the data model by implementing simple Java classes which reference each other. In most cases, this is however not enough to fully specify the mapping between those classes and their relational counterparts in the database. JPA handles this by allowing special annotations to the classes and their fields, which, for example, specify things like multiplicity and direction of relations between classes and whether a field is nullable (i.e. optional) or not.[1]

Here is an example of an annotated class, fully specifying its entity, in this case a lecture:

```
@Entity
public class Lecture implements Serializable {

        private static final long serialVersionUID = 1L;

        @Id
        @GeneratedValue
        private Long id;

        @NotNull
        private String name;

        @ManyToOne @NotNull
        private University university;

        @Embedded
        private Semester semester;

        @ManyToOne @NotNull
        private Lecturer lecturer;

        @OneToMany(mappedBy="lecture")
        private List<Attendance> attendance;

        public Long getId() {
                return id;
        }
```

[1]There is also the possibility to specify the mapping in auxiliary XML-files, but we chose the newer approach of using annotations to keep the amount of boilerplate code low.

```
        public void setId(Long id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public Lecturer getLecturer() {
                return lecturer;
        }

        public void setLecturer(Lecturer lecturer) {
                this.lecturer = lecturer;
        }

    ...
    /* more property getters and setters here */
    ...

        @Transient
        public Long getPeriodSortKey() {
                return (semester.getYear()*10) + (semester.getSummer() ? 0 : 1);
        }

}
```

Note that while most methods are simple getters and setters for the fields, some additional convenience methods may be present (here `getPeriodSortKey()`, which returns a key useful for sorting lectures by when they occurred). Such methods are marked with the `@Transient` annotation, to indicate that they aren't relevant to the data model's structure. This is not strictly necessary because we let the fields govern an entity's structure instead of their getters and setters (a choice available in JPA), but we chose to use this annotation anyway for documentational purposes.

Besides pure getters/setters and the aforementioned trivial convenience methods, no modification of state or computation is done in any of the entity classes, so as to keep the data model and any application logic strictly separate.

See figure 6.1 for a complete UML diagram of the classes which implement the data model. Note that this is *not* an entity-relationship diagram but a class diagram with associations. The multiplicities shown are between single *instances*.

The resulting database tables that Hibernate creates are presented in 6.2.

6.1.2. Structural Decisions

Most entities arise naturally by ensuring that the data model remains in 4th normal form. Universities, Lecturers, Students, Lectures and Transcripts are directly represented by their

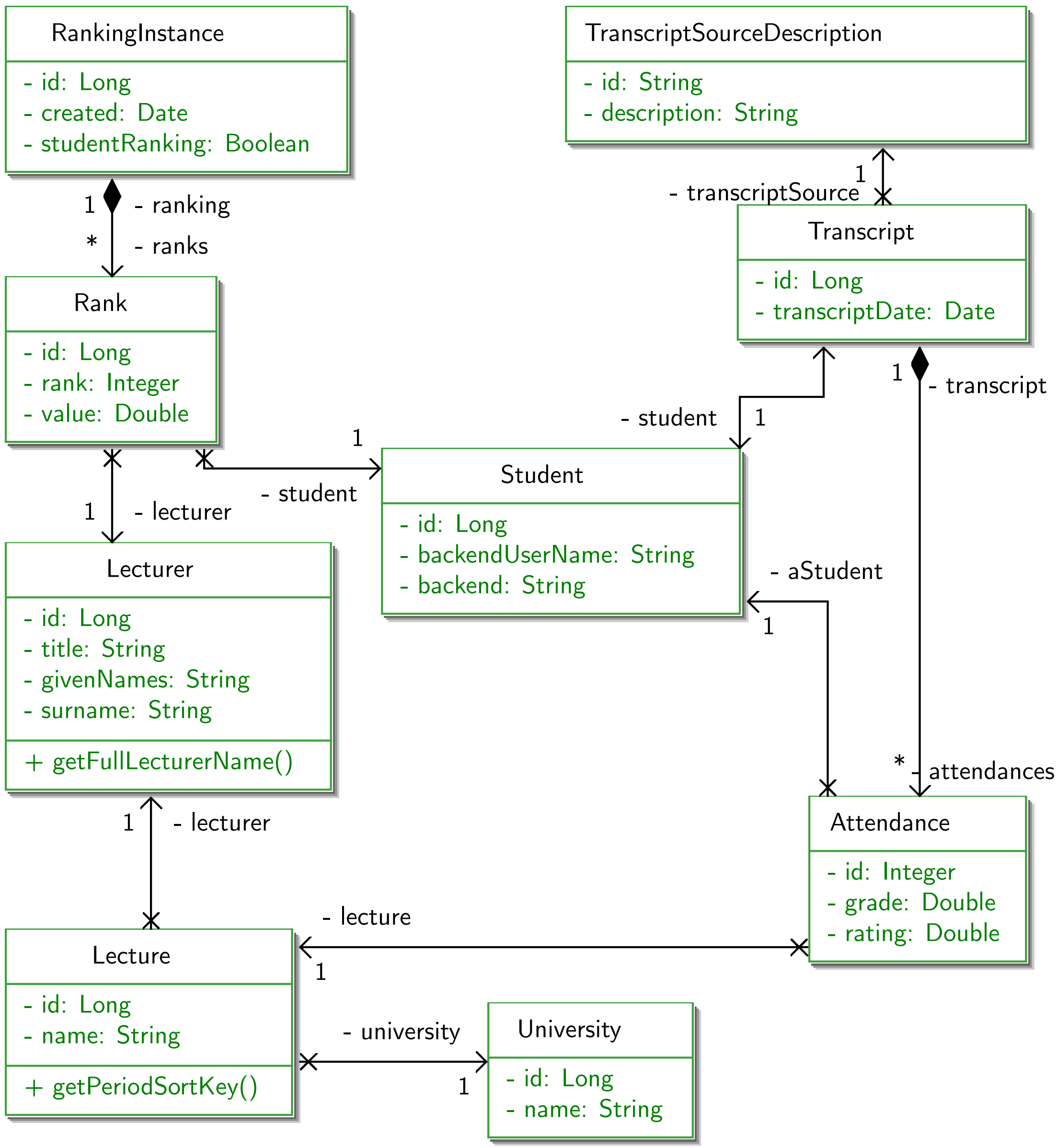


Figure 6.1.: Class diagram of the Java classes making up the data model

respective entities. They often have obvious relationships, also directly modeled by appropriate references. We will therefore focus on the non-obvious choices made for certain entities and their relationships.

Students and Their Universities

One question arising was whether to model a direct relationship between student and universities. At first glance, one might think that a student is indeed enrolled in exactly one university, or if he is not, that the attendance of multiple universities pertains to equally multiple educational goals, making each set of activities for a given goal at one institution disjoint from the respective other.

Under this viewpoint, a student that, for example, studies Physics at LMU and Electrical

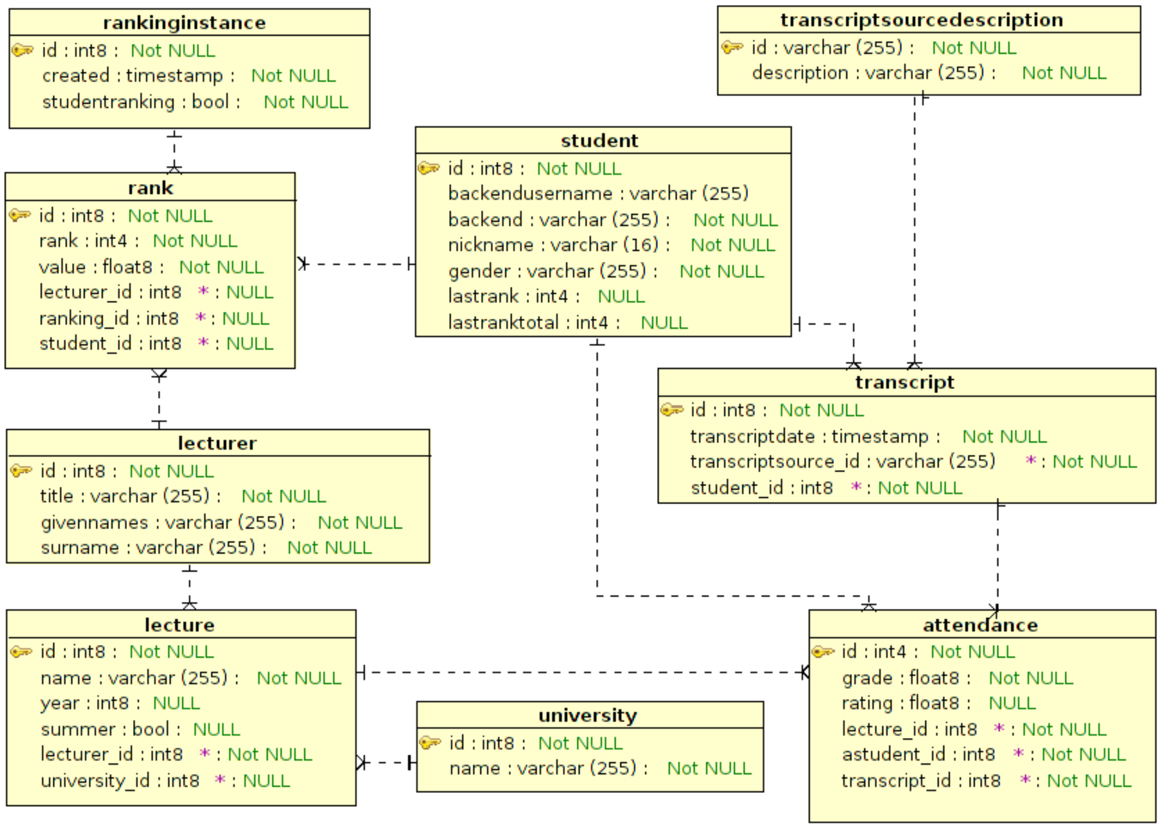


Figure 6.2.: Database schema created by Hibernate

Engineering at TU, could then be seen as two separate incarnations of a student entity: The person in their role as a Physics student, and the person in their role as an Electrical Engineering student, resulting in two accounts within our system. Since both sets of activity (their attendance of lectures) are essentially disjoint, giving a person the opportunity to rate lectures in those two separate roles respectively makes sense. (Note that a student may also enroll for two separate majors at the same university–which, in this scenario, would still mean two accounts, just however pointing to the same university.)

However, closer examination of the German university system reveals that not every student's activity at one university is attributable to the educational goal within the same university. Indeed, for some majors, German students are actually required to enroll in two universities (and attend lectures at both), despite their following of just one degree. The *Bioinformatics* course which results in a bachelor's degree, and which consists of lectures that are jointly held at TU *and* LMU is an example of that. Students participating in that course even receive two separate matriculation numbers from both universities, yet only one degree.

For this reason, we decided against a direct relationship between student and university. Instead, each *lecture* pertains to one university in our data model. Students attend any lecture at any university and, if and only if the attendance is acknowledged by its presence in the student's official transcript, are allowed to rate it.

Note that, while lectures targeted to students of several different universities do exist, they can always be attributed to one institute by which it is officially held. But the fact that students are not tied to any university (and by extension, its pool of available lectures) at all handles these cases gracefully.

Identification of Students

Although students are not tied to individual universities with respect to the attendance and rating of lectures, there is a relationship when it comes to the identification (and authentication) of a student.

We do not want to manually create and maintain student accounts, nor do we want to allow anyone to create an account into the system without (implicit) approval of the user's student status. Instead, some system of a university should provide a student's existence and authentication as a backend. If a new student, i.e. a student that has not yet logged in to our Web application, passes that backend's authentication, we want the Web application to implicitly create an account and tie the backend's student identity as well as all our locally created data to it.

Thus, the Student entity of our data model, which represents a student's account, references a student's identity as governed by the respective backend via its fields. However, this is not a normal reference to another entity within our data model. Multiple backends may exist, each interfacing a system which is not part of our application, and the representation of what constitutes a student's identity may differ vastly among those backends.

For that reason, the student's identity is only loosely referenced in its entity via a tuple of two strings: a *backend name*, which specifies which backend the student identifies and authenticates through, and a backend specific *backend username*.

The backend's application code is responsible for providing and later making sense of the backend username, whereas this backend code is chosen by the backend name. This very loose association between Student entity and the identity providing backend allows the seamless incorporation of new co-existing university backends at any time, without any change to the data model.

A caveat of this approach is the problem that a single student may have identifications at several backends, and thus create multiple Web application accounts using these different (backend, backend username) tuples, which, as explained in the previous section, causes problems when those per-backend identities aren't pertaining to disjoint courses. For example, Bioinformatics students may have email accounts at both LMU and TU and, provided that there are backends to both of those systems in the Web application, may create two accounts.

Currently, the application uses only one backend, so addressing this problem was not yet a necessity. For new backends in the future, however, we propose the following approaches:

1. Backends could check whether they identify a student which already owns a Web application account using another backend. For this to work, backends would need to be aware of each other and communicate to some extent. However, how to find out whether two separate accounts from two separate backends identify the same person is non-trivial and probably only feasible when multiple backends of only one university are in play, providing the availability of unique identifiers such as matriculation numbers.

2. Assuming that transcripts are issued only once by only one institution or institution-spanning system, the problem can be solved by allowing students to import each transcript only once *across the whole system*. This makes each attendance of any lecture tied to one single account and thus only one rating possible.

Multiplicity Between Lecturer and Lecture

The relationship between lecturers and their lectures may be modeled with two different choices of multiplicity:

1 : n A lecturer is involved with many lectures, but each lecture is only held by exactly one lecturer.

n : m Each lecture may be held as a joint effort of arbitrarily many lecturers.

n : m is favorable from a flexibility viewpoint. However, each lecture at LMU can presently be attributed, even if held by multiple lecturers, to one single authoritative lecturer. Because the information from the data sources are also presented in that way, the simpler and less elaborate 1 : n variant has been chosen, although changing the data model to accommodate an n : m approach is not considered a particularly laborious task.

Attendance

Any number of students attend any number of lectures. While this is an n : m relationship and could be modeled accordingly by just using a `ManyToMany` mapping between the Student and Lecture entities, we need to include additional information attached to the attendance itself. We therefore created an explicit Attendance entity which sits in the middle between Student and Lecture.

The Attendance entity contains the grade that a student received in one direction and the rating that a student would like to give back to the lecture in the other direction as attributes of the student's particular attendance. This rating is optional, as a student may not have given any rating yet, and may even not plan to do so. The grade, however, is mandatory, as a student requires it to be able to rate a lecture.

Additionally, the attendance is established by its inclusion in an imported transcript. To allow transcript revocations to cascadely delete the associated attendance records we include a relationship to the Transcript entity as a field of the Attendance entity.

Of note is the `aStudent` field, which creates the relationship between Student and Attendance. Because an attendance belongs to one transcript and a transcript belongs to one student, the `aStudent` field is not strictly necessary. The student belonging to an attendance is already specified indirectly via its originating transcript. In fact, including this field in the Attendance entity is a violation of normal form.

However, queries involving attendances of specific students are very common. To avoid complicated joins between Attendance, Transcript and Student in all of those queries and potential performance problems, we chose to include the `aStudent` field directly in the Attendance entity. As this field is only set once as part of the creation of any Attendance instance, it imposes no additional maintenance care or consistency hazards.

6.2. Seam Components

Seam *components* are the basic building blocks of Seam Applications. They are essentially POJOs (Plain Old Java Objects, as described for example by Richardson [28]) or EJBs (more on that later) which are instantiated by the Seam Framework itself and subsequently available through *JSF-EL* expressions for views and by being *injected* into other Seam components for the business logic. We will explain in general what this entails and then take a detailed look at each of the Web platform's implemented components.

6.2.1. Creating a Component

From a source code point of view, a POJO component is first and foremost just a simple Java class with some added annotations. No special interfaces need to be implemented, nor do they need to extend any particular classes. To make instances of such a class available through the Seam Framework, it suffices to add Seam's `@Name` annotation. For example, we would like to have a *Transcript Manager* available in our application, through which a user can browse and revoke his or her transcripts:

```
@Name("transcriptManager")
public class TranscriptManagerBean {
  ...
}
```

Instances of the component are then available through the specified name.

Local Business Interfaces

While we sometimes do use Plain Old Java Objects as Seam components (for example the `RankingProcessor`), there are a lot of cases where we would like to use features of the EJB container that the application resides in. Most prominently we would like to have access to the container's `PersistenceContext`, which gives us transactional access to the container's persistence layer (in simple words: it gives us the ability to inject an object into the component, through which we can access the database, and makes sure that everything that happens to the database within the invocation of an action method is safely wrapped in a transaction).

This means that instead of creating simple POJOs, we need to turn the classes into *Enterprise Java Beans* according to the EJB 3.0 specification (see Burke and Monson-Haefel [10]). Luckily, EJBs corresponding to that standard are not too different from simple POJOs on the surface. The only requirement for a class to be usable as an EJB, and therefore have access to EJB-specific features, is that it implements a *local business interface*, that simply contains the methods that should be available for invocation from View Facelets or other components.

For the Transcript Manager example, the full business interface is declared like this:

```
@Local
public interface TranscriptManager
{
        public void browseTranscripts();
        public void revokeTranscript();
}
```

And the actual component just implements it:

```
@Name("transcriptManager")
public class TranscriptManagerBean implements TranscriptManager {
  ...
}
```

The requirement to first declare a business interface for Enterprise Java Beans is removed in EJB 3.1.

6.2.2. Contexts

Seam manages the full lifetime cycle of components, from instantiation to destruction. It creates an instance of a component within a specific *context*, which determines the scope in which an individual instance is available, and how instances are potentially shared across the application.[2]

The great benefit is that we do not need to manage our session or application context ourselves. If we want to make an object available throughout the application, or per user session or even just within a single request, we just specify the context of that component accordingly. This is commonly done by using the `@Scope` annotation to the classes definition, as demonstrated here in the definition of the Profile Manager (which is another actual component of the Web application that gives us access to the currently logged in student's profile data):

```
@Name("profileManager")
@Scope(ScopeType.SESSION)
@AutoCreate
public class ProfileManagerBean implements ProfileManager {
    ...
}
```

The `@AutoCreate` annotations tells Seam that it should create an instance of the component itself if it is not yet available within the corresponding scope.

Seam supports a wide variety of contexts. The ones that are relevant to our Web application, ordered from widest to narrowest, are:

APPLICATION The context with the widest scope. An object that resides within this scope is available as a single instance from everywhere in the application. Consequently, it is only suitable for state that pertains to every user.

SESSION One of the most commonly used contexts. Instances for components that reside within this scope are created once per browser session. Useful for state that belongs to the single session of a single user.

CONVERSATION The conversation context is a new type of context introduced by Seam and lies between the session context and the page context. It gives the ability to divide a single session into multiple request spanning, possibly overlapping *conversations*. It is suitable for tasks that span multiple requests in the browser and allows a user to, for example, conduct multiple of these tasks at ones in different browser tabs. Though conversations are a very useful concept, they are rarely needed in our Web application.

PAGE Any object living in this context has an instance per *rendered page*. Note that this may still span multiple requests initiated by a user on that page after it has been rendered, but all of these requests pertain to, for example, additional AJAX-type requests of one open instance of that page.

EVENT The event context is for components whose life time should not exceed the scope of a single *request*.

[2] All this is also available through EJB, but Seam provides additional features and contexts.

STATELESS The narrowest "scope". Stateless objects are basically only useful for methods that do not wish to keep any state whatsoever.

6.2.3. Injection

After being created within a context, a suitable instance of a component is then available through injection. For example, the code above makes sure that a Profile Manager is created in the session context, and we can make it available in the Transcript Manager by letting Seam inject the Profile Manager. Enabling injection of a Profile Manager is as simple as declaring it as an attribute in the Transcript Manager's class and supplementing the declaration with Seam's `@In` annotation:

```
@Name("transcriptManager")
public class TranscriptManagerBean implements TranscriptManager
{
        ...
        @In private ProfileManager profileManager;
    ...
}
```

Here, the name of the attribute is the same as the Profile Manager component's name (given by its `@Name` annotation), so we do not need to explicitly specify it.

Because the Profile Manager is in the *session* scope, Seam will create an own instance of the Profile Manager within any session. This makes sense, since it contains state that is only pertaining to the currently logged in user.

JSF-EL

Within JSF, i.e. within pages of the application's views, components are directly accessible through JSF expression.

As an example, accessing the Profile Manager's `currentStudent` property (available through its getters and setters, `getCurrentStudent` and `setCurrentStudent` respectively) is simply achieved by writing:

```
#{profileManager.currentStudent}
```

6.2.4. Outjection

The opposite of injection is outjection. Attributes that are annotated with `@Out` are outjected into the appropriate context when invocation ends, meaning that to replace an object in its particular context it is sufficient to change the component's attribute within an invoked method.

```
...
public class ProfileManagerBean implements ProfileManager {
    ...
        @Out(required=false) private Student currentStudent;
    ...
}
```

The combination of injection and outjection is what Seam calls *bijection*.

Outjection is seldom used in `eval`, as it is often conveniently replaced by properties with public getters (and, if appropriate, setters), with the added benefit of not cluttering the context namespaces. In fact, later versions of Seam do not support outjection anymore, which is why we prefer to avoid it.

6.3. Components of `eval`

Appendix A describes the components implemented within the Web Application. For every component we give its name, its scope, a description of its general function and an explanation of every public method it implements. Getters and Setters for properties are omitted, instead the properties are described separately after the methods.

Some methods are restricted to logged in students or admins, which is enforced by annotating them with `@Restrict("#{s:hasRole('student')}")` (or `'admin'` respectively), a feature offered by Seam Security. If such a restriction is in place, it is explicitly mentioned in the method's description.

Because a lot of the components directly implement the actions of a particular view, we also specify the name of such associated views in these cases. Those are effectively the "Models" of MVC. All available views, including the ones that have no direct association with any component, will be explained in detail in section 6.4.

Additional components which pertain to `eval`'s deployed prototype only are described in their own Appendix B, also see section 7.3.

6.4. Views

Views define what the user actually sees and interacts with, meaning that they essentially provide the user interface of the Web Application.

Seam enhances the JSF [3] Web framework and uses Facelets [19] to render the views of the Web application (other JSF-compatible view handler technologies are theoretically possible, but not recommended, as Seam provides some additional capabilities which aid the developer). The templates used by Facelets are XHTML documents with some additional XML namespaces through which the actual template functionality is provided.

Though the XHTML templates that are part of `eval` fulfill a variety of functions (amongst them layouting and tag libraries), we will hereby focus on those templates that correspond to Web pages rendered in the user's browser, as they effectively describe the user interface and as such are what best fits the term "view".

For each view, Appendix C gives the filename of its XHTML template, a general description and, if applicable, the name of the component (effectively the view's controller) that is conceptually tied to the template. We also specify any existing security restrictions, the URL path relative to the Web Application's root through which they are reached, any mapped parameters and any navigation rules, all things that are specified in the Web Application's `pages.xml`.

As with the data model extensions and components before, views that are exclusively part of the prototype are described in their own Appendix D, with a general overview in section 7.4.

7. Prototype of `eval`

Because the bipartite algorithm depends on the availability of students' grades to calculate a ranking, `eval` has been designed with official institute backends in mind. In such a scenario, the Web Application would be fed grades by authoritative systems provided by the various institutions themselves, once a student agrees with the import of their data.

As such, it becomes subject to elaborate security implications and must be maintained carefully, as it is not possible to fully decouple the students' grades from their backend identities. The rating graph needs the individual students' grades complementary to the individual students' ratings and students must only be allowed to grade lectures they officially attended, so the grades cannot be fully anonymized.

Conclusively, if attackers would somehow gain access to the database, regardless of the channel through which such an attack took place, they would effectively gain access to all student grades that are currently residing in the system as imported transcripts.

Note that it is by far not enough to secure the Web Application *itself* (i.e. its actual code, its framework etc.) against attacks. The network, the server operating systems and the database server need to be guarded as well, among still other attack vectors that are far too numerous and complex to fit in the scope of this thesis. Moreover, due to, for example, various journaling techniques and backups, all things that are handled transparently to the Web Application, already *revoked* data could in theory be recoverable by malicious attackers.

For official systems which are part of the institutions themselves, those security considerations are common and well-known. After all, the systems already contain sensible student records which must be guarded against potential threats.

Within the scope of this thesis however, we are not yet able to benefit from any "officialty" of our system. While elevating the platform to such an official status is eventually intended by the Teaching and Research Unit under which this project is being supervised, this is an elaborate task in itself for which a student cannot take responsibility.

One approach to ameliorate the situation is decoupling the *authentication backend* identity to any actual personal identity. For example, the backend can provide the Web Application with just an anonymous, uniquely generated (for use with the Web Application) numerical ID that gives no clue about the student's identity itself. Authentication itself is then also delegable to such an authoritative system on part of the institution, for example by passing authentication tokens.

This approach would indeed provide further security and is certainly recommended when elevating the project to an official status, but was not yet available to us.

In order to nevertheless roll out the platform in a live version and let students test it in the environment of our institute, an alternative authentication scheme has been devised, that allows students to remain virtually as anonymous as desired. Only an email address is required, but except for its required validity, meaning that the user needs to be able to actually receive mail to fully register, no further restriction is imposed on that address.

7.1. Authentication in the Prototype

In the prototype, users are simply managed locally as part of the same persistence layer that manages all other data. Two simple additional entities serve as storage:

- `ProtoUser`, which represents an actual user and
- `ProtoRole`, which represents a user role, of which a user may have several (currently, `student` and `admin` are relevant).

A new user registers with a username, an email address and a self-chosen password (stored one-way encrypted and with separate salt). After registration, the account is however not yet activated. Instead, a confirmation mail is sent to the user's specified email address, which contains a link that activates the user by means of a randomly generated activation code.

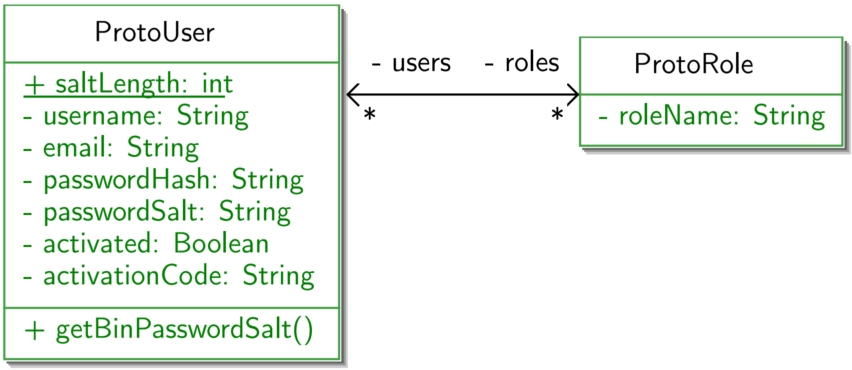


Figure 7.1.: Entity Classes for Prototype Authentication

7.2. Transcript Import in the Prototype

The prototype does not gather transcript data through an API but relies on the user uploading transcripts as XML files. This section describes those transcript XML files and details the process to import them into the Web Application.

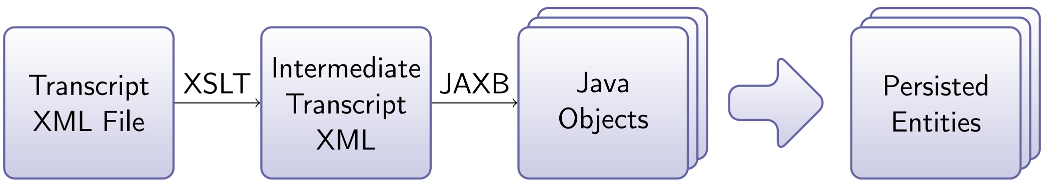


Figure 7.2.: Schematic of Transcript Import

7.2.1. Transcript XML Files

The structure of the XML files has been predetermined by the Department for Informatics at LMU. Due to the nature of the availability of transcripts in XML format as an interim solution, there is no specification (e.g. as DTD, XML-Schema or in textual form). Instead, we have been given anonymized example files that allowed us to extrapolate the structure to a degree sufficient for import.

The following is an extract of such an exemplary transcript XML file:[1]

```
<leistungen>
<name><![CDATA[Mustermann, Max]]></name>
<geburtsdaten><![CDATA[29.01.1985 in München]]></geburtsdaten>
<matrikelnummer><![CDATA[777777777]]></matrikelnummer>
<ident><![CDATA[D-2011-10-17-T-14:24:40-K-43845679]]></ident>
<kopf_ende/>

<leistung>
<veranstaltung><![CDATA[Exemplary Lecture I]]></veranstaltung>
<veranstaltung><![CDATA[Exemplary Lecture 1]]></veranstaltung>
<veranstaltung><![CDATA[Examples: Exemplary Lecture I]]></veranstaltung>
<veranstalter><![CDATA[Mustermann]]></veranstalter>
<fach><![CDATA[Exemplary Studies]]></fach>
<semester><![CDATA[SS 08]]></semester>
<ects><![CDATA[6]]></ects>
<sws><![CDATA[2+2]]></sws>
<typ><![CDATA[Vorlesung & Übung]]></typ>
<termin><![CDATA[]]></termin>
<pruefungsinstanz><![CDATA[]]></pruefungsinstanz>
<note><![CDATA[2.70]]></note>
</leistung>

<leistung>
<veranstaltung><![CDATA[History of Examples]]></veranstaltung>
<veranstalter><![CDATA[Musterfrau]]></veranstalter>
<fach><![CDATA[Informatik]]></fach>
<semester><![CDATA[WS 08/09]]></semester>
<ects><![CDATA[6]]></ects>
<sws><![CDATA[3+2]]></sws>
<typ><![CDATA[Vorlesung & Übung]]></typ>
<pruefungsinstanz><![CDATA[1]]></pruefungsinstanz>
<note><![CDATA[4,0]]></note>
</leistung>

...

</leistung>
</leistungen>
```

We can informally describe transcript XML files received from the Department of Informatics as follows. Since the element names are in German, we provide an English translation for each mentioned element in parentheses (note that the translation is not necessarily exact, but tailored to its context).

- The root element of the document is `leistungen` (*performances*).
- The first elements form a "header" which identify student and transcript. This "header" ends with the delimiting element `kopf_ende` (*end_of_header*). As described later, all of

[1] further modified for clarity and anonymization.

this header information will be discarded.

- The rest of the document consists of a sequence of `leistung` (*performance*) elements, which denote the individual performances of a student's attendances.
- Each performance contains a multitude of elements, some apparently optional, that describe the lecture as well as the performance of the student in the lecture's examination.

  Of those elements, only the following will be relevant for import:

  - `veranstaltung` (*lecture*): The name of the lecture. This element can appear multiple times with variations of the lecture's name.
  - `veranstalter` (*lecturer*): The surname of the lecturer giving the lecture. (Other parts of the name, such as given names or titles, are not contained in the document.)
  - `semester`: The semester in which the lecture and examination took place, in the form `SS` *`YY`* for summer semesters and `WS` *`YY/ZZ`* for winter semesters.
  - `note` (*grade*): The grade that the student received in the lecture's examination, corresponding to the German university grade system. Note that the format is inconsistent across instances of performances: In the first performance, a decimal point is used whereas in the second performance, a comma is used as decimal delimiter (as common in Germany).

Upon inspection, there are some problems with straightforward processing of such documents. While the inconsistency in the numeric format for grades is easily remedied by replacing each comma in a grad with a dot character, the lack of any formal specification of the document's overall structure is of greater concern.

For example, within a `leistung` element, there are no real guarantees about which elements are optional and which elements may appear in what kind of order (in the worst case, previously unseen elements may appear in future transcript XMLs).

A lack of such guarantees means that an XML-Schema that accurately describes a transcript XML file would need to be more tolerant in what it validates than is preferable for certain uses of it, lest a too rigid XML-Schema may break transcript import for transcripts that are unexpectedly different from the available examples.

One of these uses is that we would like to unmarshal the XML into Java objects by using JAXB, the *Java Architecture for XML Binding*, as this vastly simplifies the import process and appears much less awkward in code than using the DOM API to traverse the document or writing SAX event listeners. However, in order to generate Java classes by the JAXB tools and to really benefit from the unmarshalling, an appropriately rigid XML-Schema is needed (we will describe the JAXB step later on).

To remedy this situation, the transcript XML document is easily translated into another document with a more well-defined structure by performing an XSL transformation. This step is justified by the fact that the XSLT stylesheet that tolerantly performs all cleanup and restructuring is fairly short and simple, and that incorporating the transformation step is a matter of a few simple lines in the Java application logic code, since XSL transformers are readily available as part of Java EE.

The added benefit of the intermediate document is that its newly defined document structure is not directly tied to the original document structure of prototype transcript XML files, and can be used for other, future university backends that use XML (possibly through APIs instead of file upload) as well. In such cases, the programmer only needs to write new XSLT stylesheets that transform the source XML into the described document structure, while the Java transcript import code itself may remain untouched (if the extent of its functionality suffices).

Note that the JAXB unmarshaller can be directly fed with the output destination object of the XSL transformer, which averts any explicit buffering of the intermediate document in the source code and further simplifies things.

7.2.2. Intermediate Transcript Document Structure

Figure 7.3 shows the XML-Schema definition of the intermediate document. An example gives a more intuitive impression of such a document:

```
<grades>
  <lecture>
    <names>
      <name>Exemplary Lecture I</name>
      <name>Exemplary Lecture 1</name>
      <name>Examples: Exemplary Lecture I</name>
    </names>
    <lecturer>Mustermann</lecturer>
    <semester>
      <year>2008</year>
      <summer>true</summer>
    </semester>
    <grade>2.7</grade>
  </lecture>

  <lecture>
    <names>
      <name>History of Examples</name>
    </names>
    <lecturer>Musterfrau</lecturer>
    <semester>
      <year>2008</year>
      <summer>false</summer>
    </semester>
    <grade>4.0</grade>
  </lecture>
</grades>
```

The differences between the original format and the newly defined format can be summarized as follows:

- The “header” elements under the root elements are not included, therefore the root element `grades` only contains a sequence of performances (now called `lectures`).
- In the individual lectures, only the names, the name of the lecturer, the semester and the received grade are taken over. All corresponding elements must be present and in a specific order.
- The sequence of lecture names is now under its own element, `name`.

- The semester is now a sequence itself, consisting of two elements with rigidly defined data types: `year`, which is defined as a Gregorian year and consequently contains the full year instead of the last two digits, and `summer`, which is a boolean that indicates a summer semester when `true` and a winter semester when `false`. This form makes the specification of a semester more well-defined and much easier to handle in every subsequent processing.
- The `grade` is now of data type decimal, and contents of this element are formatted accordingly.

It is clear that this XML is easier to handle in practically every regard, not because of the omission of many elements (if needed, those could be included in the schema and consequently in documents without any disadvantages), but because of the rigidly defined structure and unambiguous datatypes. As already mentioned, the same or an extended version of this XML-Schema can also be used for other transcript backends.

Figure 7.4 shows the stylesheet which performs the transformation from the original to the newly defined format.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="grades">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="lecture"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="lecture">
    <xs:complexType>
      <xs:all>
        <xs:element ref="names"/>
        <xs:element ref="lecturer"/>
        <xs:element ref="semester"/>
        <xs:element ref="grade"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="names">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="name"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="semester">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="year" type="xs:gYear"/>
        <xs:element name="summer" type="xs:boolean"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="lecturer" type="xs:string"/>
  <xs:element name="grade" type="xs:decimal"/>
</xs:schema>
```

Figure 7.3.: grades.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" media-type="text/xml"/>

  <xsl:template match="/leistungen">
    <grades>
      <xsl:apply-templates select="leistung"/>
    </grades>
  </xsl:template>

  <!-- Extract lectures, but only those that have an assigned lecturer and
     a grade. (See the grade extraction below for an explanation of the
       translate expression.) -->
  <xsl:template
    match="leistung[string-length(veranstalter)
␣␣␣␣␣␣␣␣␣␣␣&gt;␣0␣and␣number(translate(note,',','.'))␣&gt;␣0]">
    <lecture>
      <!-- Extract all lecture names. -->
      <names>
        <xsl:for-each select="veranstaltung">
          <name><xsl:value-of select="."/></name>
        </xsl:for-each>
      </names>

      <!-- Extract the lecturer. We extract the first one, assuming that
           he or she is the responsible lecturer, in the case that there
           are multiple elements (there is not in the sample XML, but lack
           of any official specification makes us want to play safe). -->
      <lecturer><xsl:value-of select="veranstalter[1]"/></lecturer>

      <!-- Extract the semester, breaking down the compound string. -->
      <semester>
        <year>
          <xsl:value-of select="concat('20',substring(semester,4,2))"/>
        </year>
        <xsl:choose>
          <xsl:when test="substring(semester,1,1)␣=␣'S'">
            <summer>true</summer>
          </xsl:when>
          <xsl:otherwise><summer>false</summer></xsl:otherwise>
        </xsl:choose>
      </semester>

      <!-- Extract the grade. Note that the decimal mark is inconsistent in
           the sample XML: sometimes a dot (.) is used, sometimes a comma (,).
           We therefore replace commas with dots, as that is what the XPath
           engine expects. -->
      <grade><xsl:value-of select="number(translate(note,',','.'))"/></grade>
    </lecture>
  </xsl:template>

  <!-- Ignore ill-fitting lectures (no name or no grade). -->
  <xsl:template match="leistung"/>

</xsl:stylesheet>
```

Figure 7.4.: `extract-grades.xsl`

7.2.3. Unmarshalling via JAXB

With the XML-Schema definition of the intermediate document form, Java classes are generated. By calling the JAXB schema compiler with

```
xjc -p bipartit.proto.jaxb grades.xsd
```

the following classes are generated in the `bipartit.proto.jaxb` package:

- `ObjectFactory`: used to programatically construct new instances, not directly used in the Web Application.
- `Grades`: object representation of the document's root element, contains a `List` of `Lectures`.
- `Lecture`: object representation of a lecture, contains the grade (as `BigDecimal`) and lecturer name (`String`), as well as references to `Names` and `Semester`.
- `Names`: a `List` of `String` names, the names of the lecture.
- `Semester`: Object representation of a semester.

To actually unmarshal a (transformed) document into this simple object representation, we first set up the unmarshaller:

```
JAXBResult result = new JAXBResult(
                        JAXBContext.newInstance("bipartit.proto.jaxb"));
```

And can then directly instruct the XSL transformer to stream its output into the unmarshaller, where, after successful completion, an instance of class `Grades` contains the intermediate document in its object representation:

```
transcriptTransformer.transform(source, result);

grades = (bipartit.proto.jaxb.Grades) result.getResult();
```

From then on, all XML-related processing is finished and the `grades` objects is used to import attendances as persistent entities.

7.2.4. Importing Attendances as Entities

With the transcript structure available in the aforementioned hierarchy, importing the attendances as entities into the database is fairly simple:

1. A new `Transcript` entity is created, and its metadata (owning student, current date as transcript date) is set. We use `"PROTO"` as backend identifier.
2. For each of the `Lectures`, i.e. each of the attendance records in the transcript, the following steps are performed:
   a) A `Lecturer` entity is looked up, whose surname matches the lecturer's surname in the attendance record. If none is found, a new one is created and persisted into the database.
   b) A `Lecture` entity is looked up, for which one of the names in the attendance record matches the lecture's name in the database, as well as lecturer (matching the `Lecturer` entity in the previous step) and semester. If none exists, a new one is

created and persisted into the database. Note that we only save one name per lecture (in this case, the first one that is encountered).

c) A new attendance record that points to the previously found or created `Lecture` entity is created under the new transcript. Grades are set accordingly.

The transcript and all of is dependencies are now persisted into the database.

7.3. Components

Appendix B describes additional components that are exclusive to the prototype. They include components that pertain to local entity based user registration, activation and authentication, as well as a transcript importer component, which processes and imports the transcript XML files.

7.4. Views

Besides a few extra views that pertain to local user registration and activation, prototype transcript import hooks into `transcripts.xhtml`.

All prototype-specific views have a filename prefixed with `proto-`. They are described in Appendix D.

8. Analysis Of eval²Rank

The prototype went online to kick off an institute-wide beta test on October 26, 2011. From then on, students were able to use the software in its entirety. In particular, they were able to upload transcripts, rate visited lectures and rankings were generated each night.

The following analysis was done with a copy of the database snapshotted on November 14, 2011. At that time, there were 16 students registered. 13 of these students uploaded transcripts, each student exactly one transcript. The 13 transcripts contained 209 attendances to 68 lectures. However, only 9 students actually rated any lectures. By these students, 122 ratings were given out, rating 32 of the 37 distinct lecturers. This allows us to do a first brief evaluation with real data.

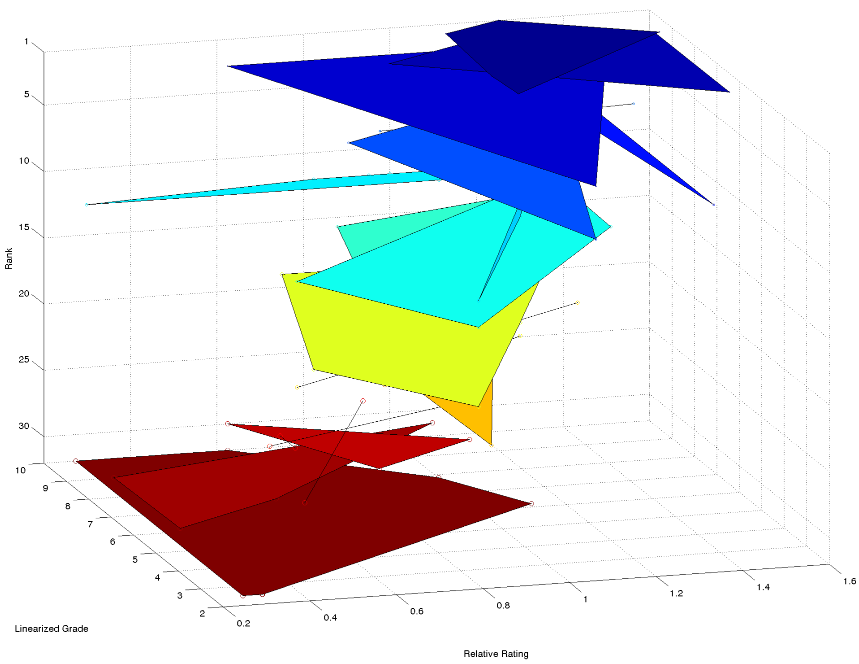


Figure 8.1.: Visualization of ratings and grades within ranks

A first visualization of the gathered data is given in figure 8.1. Each horizontal plane represents a lecturer rank, with the highest lecturer rank being on top and the lowest on bottom. Within the horizontal planes of the ranks, each relative rating given by a student forms the x-coordinate of a point, and the average grade given to the student by the lecturer belonging to that rank forms the point's y-coordinate. The convex hull of all points within the rank-representing planes is then taken and visualized as a colored patch. The patch therefore represents the relationships between received grade and given rating among the ranks.

This diagram shows how differently graded students have rated the grading lecturers. However, it does not tell much about the algorithms validity. While it is visible that, unsurprisingly, higher rated lecturers, i.e. lecturers whose surfaces tend to the right, seem to float to the top ranks, not much else which is directly visible influences the ranking. In fact, each of the patches represents the ratings and grades between students and a *single* lecturer, and we do not expect this to have too much influence in the overall ranking, as explained in the motivation for using a PageRank-based algorithm in section 3.3.

A better approach for a first evaluation of the algorithm's performance is by comparing it to another much simpler and intuitive algorithm, and explore the differences, to understand how they make sense.

Comparison of two Ranking Algorithms

Figure 8.2 shows the most current ranking of the (anonymized) lecturers on the left side, using the implemented eval²Rank algorithm. The table on the right side shows the outcome of the trivial Average Rating (AR) ranking algorithm, that ranks lecturers by their overall average rating given out by all students that rated them. In this simple algorithm, the students' grades aren't regarded at all. Lines between the left and right sub-tables connect the same lecturers in their respective ranking, to visualize the differences in rank.

As expected, the rankings are not radically different, i.e. the differences are mostly local. The lines that connect the two tables form some parallel "clusters", which indicate that the differences there are mostly from comparably few lecturers changing place. It is important to note, though, that eval²Rank, due to the additional information it is able to use, generates more distinct ranks than the AR algorithm. Because of that, it is not always possible to say whether a lecturer would have switched places if it were on a distinct rank to begin with, e.g. due to more ratings that give more distinct averages.

None of the lecturers change their rank by more than 8 places, which is ¼th of the whole field: From the AR ranking to the eval²Rank ranking (from right to left in the figure), the now top positioned *Lecturer345* rises the highest by 7 places (from 8 to 1) and *Lecturer556*, while the former top lecturer in the AR ranking falls the lowest, by 8 places from 1 to 9.

We will look at *Lecturer556*'s fall first, because it appears most grave. From the individual ratings, we see that this lecturer has only been rated once.[1] The student that rated this lecture has an average grade of 6.44 points (out of the internal 10-point-scale, see 4.5.2 for the conversion), which corresponds to a grade between 3.0 and 3.3 on the German grade scale. This grade, being towards the lower end, highly suggests that the students rank will be low as well, and therefore less relevant in influence. Indeed, an examination of the student's ranking reveals that his or her rank is 9, which is the lowest rank.

By contrast, *Lecturer27*, which is the new top lecturer, has been rated 12 times in total. 10 of this 12 students have a grade of at least 8.84 points (better than 2.0 on the German grade scale), which highly indicates a high ranking and a high influence in eval²Rank.

It is therefore justifiable that the, comparatively, highly regarded *Lecturer27* takes the place of the as yet insufficiently rated *Lecturer556*.

Similarly, *Lecturer525*, *Lecturer539* and *Lecturer553*, which all share rank 13, have only received one (albeit high) grade from the same last-ranked student, therefore pushing their

[1] We will not publish the individual grades and ratings, as we feel that they contain too much information, which might be used to identify students. Therefore, this section only contains sufficiently aggregated data.

ranks down and the ranks of their original followers up.

Lecturer345, also a comparably high riser (16 to 11) is interesting, because it is not immediately visible why this lecturer rises above his or her predecessors in the AR ranking by looking at the absolute ratings only. Only three students have rated this lecturer, with a rating of 6, 8 and 8, given by students with rather low rankings of 9, 6 and 7 respectively. While some of the three original predecessors received low ratings from highly ranked students, which might explain why they have been pushed down, one lecturer, *Lecturer431*, received only a good rating from a high-ranked student.

Remember, however, that the ratings of a student as used in the algorithm are not regarded absolutely but relatively to all other ratings the student has given out. When examining the relative ratings, it turns out that the student who rated *Lecturer431* stays close to his or her average rating, while the two high-rating students of *Lecturer345* rated well above their average, therefore expressing their relative appreciation for *Lecturer345*.

The reason for *Lecturer70*'s fall is again visible at first glance: despite being rated with an 8 from the highest ranked student, he or she is pushed down by the even absolutely bad to average ratings of other high-ranked students, relatively all below average. One student, the second-highest ranked one, even gives a rating of only 0.625 of the average.

eval² Rank	Lecturer	Eigenvector Value	AR Rank	Lecturer	Avg. Rel. Rating
1	Lecturer27	1.151	1	Lecturer556	1.36
2	Lecturer33	1.105	2	Lecturer33	1.33
3	Lecturer19	1.088		Lecturer45	1.33
4	Lecturer48	1.086	4	Lecturer530	1.31
5	Lecturer45	1.050	5	Lecturer42	1.26
	Lecturer530	1.050	6	Lecturer48	1.25
7	Lecturer59	1.049		Lecturer275	1.25
8	Lecturer42	1.049	8	Lecturer27	1.23
9	Lecturer556	1.027	9	Lecturer19	1.13
10	Lecturer275	1.022		Lecturer59	1.13
11	Lecturer345	1.020	11	Lecturer290	1.1
12	Lecturer290	1.011		Lecturer431	1.1
13	Lecturer373	1.011	13	Lecturer525	1.09
14	Lecturer85	1.010		Lecturer539	1.09
15	Lecturer431	1.008		Lecturer553	1.09
16	Lecturer525	1.005	16	Lecturer345	1.07
	Lecturer539	1.005	17	Lecturer85	1.03
	Lecturer553	1.005		Lecturer373	1.03
19	Lecturer67	1.000	19	Lecturer67	0.97
20	Lecturer362	0.992		Lecturer362	0.97
21	Lecturer5	0.990		Lecturer368	0.97
22	Lecturer368	0.985	22	Lecturer5	0.96
23	Lecturer522	0.982	23	Lecturer70	0.86
	Lecturer542	0.982	24	Lecturer422	0.83
	Lecturer562	0.982	25	Lecturer522	0.82
	Lecturer667	0.982		Lecturer542	0.82
27	Lecturer422	0.963		Lecturer562	0.82
28	Lecturer339	0.963		Lecturer667	0.82
29	Lecturer299	0.954	29	Lecturer299	0.74
30	Lecturer70	0.931	30	Lecturer339	0.63
31	Lecturer436	0.883	31	Lecturer436	0.56
32	Lecturer39	0.657	32	Lecturer39	0.51

Figure 8.2.: Comparison of eval²Rank and Average Rating (AR)

9. Conclusion and Future Work

The goal of this thesis was to construct a functioning algorithm based on the weighted eigenvector centralities in bipartite graphs idea and to develop a fully usable and deployable version of a Web platform that uses this algorithm. After deployment, real data would be gathered, opening the opportunity for evaluation and further development of the proposed or other suitable algorithms. This goal has been achieved, and since then students are making active use of the platform.

In the last chapter, we also saw that eval²Rank gives similar results to the highly intuitive Average Rating algorithm, but takes into account the performances of the rating students themselves in a meaningful way. By doing this, the algorithm also eliminates some obvious artifacts of Average Rating, e.g. by not letting a lecturer with just a single excellent rating float to the top of the ranking, or preventing that a good lecturer will be pushed down by a few bad ratings of poor performing students.

In effect, this thesis is kicking off a new bipartite ranking algorithm and a full Web platform for bipartite Student-Lecturer evaluation and as such, it also opens numerous opportunities for further research. This chapter proposes some directions.

9.1. Multipartite Ranking: From eval²Rank To eval^{n}Rank

The system currently uses lectures just "by proxy", connecting the ratings between students and lecturers. All rankings between one student and one lecturer (or vice versa) are condensed into their average and subsequently fed into the algorithm, completely losing the concept of lectures in the process. Consequently, there is also no ranking of lectures.

A tripartite approach, where lectures are actual nodes in the rating graph, connected with students and lecturers respectively, is conceivable and would open interesting opportunities for a ranking of the lectures themselves *and* as a better Student-Lecturer ranking algorithm.

Such an *eval³Rank* approach in which the lectures form their own partition might be beneficial for lectures which are given by different lecturers each semester in which they occur. If, for example, a lecture is *itself* and independent from the lecturer rather unpopular, this unpopularity would not only reflect itself in the rankings of all lecturers, it would consequently also result in less impact to the lecturers' ranks.

Of course, it does not need to stop at $n = 3$, as the idea can easily be generalized to a multipartite *eval^{n}Rank* algorithm.

9.2. Multidimensional Rating Scenarios

Currently, the actual rating of individual lectures mimics the grading process and is therefore rather simple: one lecture is potentially assigned exactly one rating by each student.

More complex rating scenarios are easily construable and in fact commonly encountered with traditional methods of lecture evaluation.

For example, instead of just assigning one rating to one lecture, students could be able to rate several different qualities of a lecture independently, using multiple instances of the rating scale for each lecture.

While it is trivial to calculate multiple rankings from these individual ratings, one ranking for each quality, the structure and calculation of a single ranking that takes all qualities into account is yet unclear.

9.3. Investigation of the "Star" Scale Mapping

As mentioned in section 4.5.1, it is yet unclear whether the linear mapping of the 5-star-scale used in the user interface to the 10-point-scale used in the algorithm is satisfactory. The exploration of rating scales is an ongoing research topic in psychological and social studies and subject to controversy, as described for example by Jamieson [21] and Blaikie [5].

Full investigation of how to handle the rating scale would far exceed the domain of this thesis. However, if one wishes to alter either the mapping between the external (as used in the user interface) and internal rating scale or even the external rating scale itself, the complete decoupling of all scales detailed in section 4.5 allows one to easily do so.

9.4. Rigorous Evaluation And Analysis Of The Algorithm

Although the algorithm has been motivated and a first brief analysis in section 8 shows that its differences in ranking compared to a simple Average Rating algorithm seem highly beneficial, no rigorous analysis of the algorithm's theoretical properties have been attempted, as the scope of this thesis is mainly focused on, besides development of the algorithm, the practical aspects of a Web Platform which uses it.

We also consider it to be an important task to test the algorithm against crafted data as opposed to the real data gathered here, in order to test the algorithms behavior when faced with attempts to "game" the system, possibly by arrangements in groups which include members of both partitions. For instance, vanilla PageRank is prone to manipulation by so-called link farms, as described by e.g. Gyongyi and Garcia-Molina [16] and Du et al. [13]. Investigating how this issue translates to bipartite rankings seems to be a worthwhile endeavor.

However, the platform that is the subject of this thesis offers a live-deployed, fully extensible framework that also opens the opportunity for rigorous evaluation of the algorithm against the increasingly available data and experimentation.

A. Components

The following appendices describe the components and views of `eval` in detail. The goal is to allow future developers of `eval` to use these appendices as helpful references.

See 6.3 for an explanation of the components in this section.

A.1. `editProfile`

Class	`EditProfileBean`
Business Interface	`EditProfile`
Scope	Stateless
Associated View	`editProfile.xhtml`
Description	Provides the action for saving an edited user profile.

Methods

`void saveProfile()`	Action called when the user wishes to save their profile. This action is called after modifying the `Student` entity obtained by the property `profileManager.currentStudent` (whose properties in turn are directly bound by input fields in the `editProfile.xhtml` view). It merges the entity into the database if the nickname is not duplicate, otherwise an error is created as a status message.

A.2. `feedback`

Class	`FeedbackBean`
Business Interface	`Feedback`
Scope	Page
Associated View	`feedback.xhtml`
Description	Provides the action for sending feedback.

Submitted feedback renders the `feedbackMail.xhtml` view internally, which resides in Seam's `http://jboss.com/products/seam/mail` XML namespace and therefore sends out a mail as a result of rendering. The feedback is sent anonymously, meaning that both sender and recipient are set to a preset email address with no mention of any user's data in the feedback mail.

Methods

`void sendFeedback()`	Send the user feedback. The user feedback in property `feedback` is sent via mail by internally rendering the `feedbackMail.xhtml` view. *This method is restricted to students.* (While the feedback itself is sent anonymously, we require users to log in in order to prevent spam.)

Properties

`String feedback`	The user feedback.

A.3. `imapAuthenticator`

Class	`ImapAuthenticator`
Business Interface	`Authenticator`
Scope	Stateless
Associated View	*none*
Description	Authenticate user via IMAP.

This component authenticates students against a (hardcoded) IMAP server. It is meant for future integration and not used in the prototype, which uses the database based `protoAuthenticator` instead (see section 7.3).

To enable Seam Security to use this component as an authenticator, configure it in `components.xml` like this:

```
<security:identity
    authenticate-method="#{imapAuthenticator.authenticate}"
    remember-me="true"/>
```

Methods

`boolean authenticate()`	Authenticate the user via IMAP. The credentials (username, password) entered by the user are injected by Seam as a component with type `Credentials` and name `credentials`. This method returns `true` if authentication against the (currently hardcoded) IMAP server succeeded and false otherwise.

A.4. `lectureManager`

Class	`LectureManagerBean`
Business Interface	`LectureManager`
Scope	Session
Associated View	`browseLectures.xhtml`
Description	Actions for browsing and rating lectures in the “Rate” tab.

The Lecture Manager is the heart of the page behind the “Rate” tab, where users may browse, filter and, if logged in as students, rate lectures. It manages the filters set by the user and a `LectureDataModel`. This `LectureDataModel` performs the actual database queries and implements windowing and prefetch strategies to enable pagination of the lecture table. The `LectureDataModel` is accessible as the `rows` property, which is directly given as a data model to the view's `<rich:dataTable>`.

Methods

`void browseAllLectures()`	Action called for the browse page. This action should be called upon visiting the “Rate” page. It populates the semester filter with all semesters of imported lectures and refreshes the `LectureDataModel` if needed.
`void reloadDataModel()`	Create a new `LectureDataModel`. This action is called whenever a new `LectureDataModel` is needed. It may also be called from other components when they knowingly changed the persisted lectures, such as after importing a transcript.
`void filter()`	Set filters in data model. The view calls this action whenever one of the filters changed. It updates the filters of the data model and re-queries the database based on the changed parameters.
`void showAll()`	Shortcut filter action for the Show All filter link.
`void showOnlyUnrated()`	Shortcut filter action for the Unrated filter link.
`void showOnlyRated()`	Shortcut filter action for the Rated filter link.
`void rate()`	Rate the lecture in the current row. Rates the selected lecture with the amount of points previously set in the `pointsToSet` property. The lecture to rate is determined by the current data table row from the data model. *This method is restricted to students.*
`void removeRating()`	Remove the rating from the current row's lecture. *This method is restricted to students.*

Properties

`Integer pointsToSet`	Amount of points to rate a lecture with.

Note: When changing any of the following properties, invoke the `filter` action.

`String filterLecture`	Filter lectures with names containing this string.
`String filterLecturer`	Filter lectures whose lecturer's surname contains this string.
`List<SelectItem> filterSemesterItems` *(read-only)*	Contains the items for the semester filter's `<h:selectOneMenu>`.
`Semester filterSemester`	Filter lectures according to the chosen semester.

`String filterShowOnly`	Filter for showing only rated, unrated or all lectures. Possible values are `"rated"`, `"unrated"` or `""` respectively. Also set by the `showOnlyRated`, `showOnlyUnRated` and `showAll` actions.
`String order`	The order in which lectures should appear in the lecture table. Possible values are `"semester"` (default), `"rating"`, `"grade"`, `"lecturer"`, `"numratings"` or `"lecture"`. `"numratings"` orders the number of ratings from highest to lowest, all other orders are ascending.

A.5. `periodicRankingSchedule`

Class	`PeriodicRankingSchedule`
Business Interface	*none*
Scope	Application
Associated View	*none*
Description	Schedule periodic rankings.

This special component just contains a single method, `void schedule()`, which is annotated with `@Create` so that it is immediately executed after component instantiation. The component's class itself is application-scoped and annotated with `@AutoCreate` and `@Startup`, meaning that it is immediately instantiated when the Web Application comes up and consequently its `schedule()` method is called as well.

The `schedule()` method just calls the asynchronous `schedulePeriodicRanking()` method of the `rankingProcessor` component (which exists in that component because of issues with the Entity Manager, see there) with the current date as start date and a hardcoded schedule as its parameters, resulting in periodic generation of rankings.

The schedule is the `cron` interval string `0 0 6 * * ?`, which specifies that the action is to be performed every day at 6:00am.

A.6. `profileManager`

Class	`ProfileManagerBean`
Business Interface	`ProfileManager`
Scope	Session
Associated View	*none*
Description	Manage student profiles: fetch or create them upon login, make current student profile available through property.

Methods

`void updateCurrentStudent()`	Fetch or create the Student entity associated with the given login credentials and set the `currentStudent` property to it. This method is called whenever the login credentials change, most notably because a user logged in (after successful authentication). To ensure this, it is annotated as an `@Observer` of the `org.jboss.seam.security.loginSuccessful` event.
`void clearCurrentStudent()`	Clears the `currentStudent` property. This method is called whenever a user logs out, by being annotated as an `@Observer` of the `org.jboss.seam.security.loggedOut` event.

Properties

`Student currentStudent` *(read-only)*	The Student entity of the currently logged in student, or *null* if no student is logged in.

A.7. `rankingProcessor`

Class	`RankingProcessor`
Business Interface	*none*
Scope	Application
Associated View	*none*
Description	Generates rankings.

Methods

`void generateRanking()`	Generates rankings for students and lecturers. Both a new student and a new lecturer ranking are generated as `RankingInstance` entities. The rankings are generated according to the algorithm fully described in chapters 3 and 4.

```
@Asynchronous
QuartzTriggerHandle
schedulePeriodicRanking(@Expiration Date when, @IntervalCron String cronInter)
```

	Schedule periodic ranking. This is a special method invoked by the `periodicRankingSchedule` component. It effectively just calls `generateRanking()`, but because it is annotated as `@Asynchronous`, the method is not executed immediately. Instead, it returns a handle, and is actually performed periodically according to the `cron`-schedule `cronInter`. Scheduling starts after the date specified by parameter `when`. While, from a design viewpoint, it would be favorable to have this method as part of the `periodicRankingSchedule` component instead, this component's usage of the Entity Manager causes problems with initialization at start-up, and is therefore contained here.

A.8. showRanking

Class	`ShowRankingBean`
Business Interface	`ShowRanking`
Scope	Page
Associated View	`showRanking.xhtml`
Description	Fetches rankings for display.

Methods

`void showRanking()`	Action called for the ranking page. This action should be called upon visiting the "Ranking" page. It initializes its `rankings` properties by fetching all available lecturer rankings into it, and setting `selectedRankingsId` to the latest ranking if it was not set before (e.g. by the `ranking` URL parameter, mapped to the property in `pages.xml`). The ranking with the selected ID is then set on the `displayRanking` property, available for display through the view.
`void generateRankingNow()`	Generate a new ranking now. Usually rankings are periodically generated by the `periodicRankingSchedule` component. This action method allows a new ranking to be generated immediately. *This method is restricted to admins.*

Properties

`List<RankingInstance>` `rankings` *(read-only)*	List of available rankings.

`RankingInstance displayRanking` *(read-only)*	Currently selected ranking.
`Long selectedRankingID`	ID of ranking to fetch into `displayRanking`.

A.9. `transcriptManager`

Class	`TranscriptManagerBean`
Business Interface	`TranscriptManager`
Scope	Session
Associated View	`transcripts.xhtml`
Description	Actions for browsing and revoking transcripts. Note that this contains no means for *importing* transcripts, as importing is the responsibility of the transcript backends. In this current version, the prototype transcript import is available via the `protoTranscript` component.

Methods

`void browseTranscripts()`	Action called for the transcript page. Fetches all transcripts of the current student into the `transcripts` property. *This method is restricted to students.*
`void revokeTranscript()`	Revoke the transcript specified by the `selectedTranscript` property. *This method is restricted to students.*

Properties

`List<Transcript> transcripts` *(read-only)*	List of available transcripts for the student.
`Transcript selectedTranscript`	Transcript selected for removal.

B. Prototype Components

See 7.3 for explanation.

B.1. `protoActivator`

Class	`ProtoActivatorBean`
Business Interface	`ProtoActivator`
Scope	Stateless
Associated View	`proto-activate.xhtml`
Description	Activate a registered user.

The account confirmation sent to a newly registered user links to a URL that sets this component's `username` and `activationCode` properties and invokes its `activate` action. If `username` and `activationCode` match the user's activation data in the database, the `activated` property of the student's `ProtoUser` entity is set to `true` and a confirmation message is displayed (otherwise, an error message is shown, prompting the user to re-register).

Methods

`void activate()`	Activate the user `username` if `activationCode` matches the activation code in the database.
`boolean isSuccess()`	`true` if activation was successful.

Properties

`String username`	Username of user to be activated. Populated by the `username` URL parameter (required).
`String activationCode`	Activation code to check against database. Populated by the `code` URL parameter (required).

B.2. `protoAuthenticator`

Class	`ProtoAuthenticator`
Business Interface	`Authenticator`
Scope	Stateless
Associated View	*none*
Description	Authenticate user via prototype user entities.

This component authenticates students against the prototype user entities which are entirely contained in the database. No university backend check is performed, instead any user may register in the Web Application (through the `protoRegister` component) and, once activated via the `protoActivator` component, may login using this authenticator.

While the associated `ProtoUser` entity contains the user's email address which must be valid (otherwise the user will not receive the confirmation mail containing their activation code), the address may be *any* valid address and no domain check is performed against it. This is purposefully done to allow students to register to the prototype without revealing any personal information.

This component is enabled as an authenticator in `components.xml` as follows:

```
<security:identity
    authenticate-method="#{protoAuthenticator.authenticate}"
    remember-me="true"/>
```

Methods

`boolean authenticate()`	Authenticate the user via `ProtoUser` entity. The credentials (username, password) entered by the user are injected by Seam as a component with type `Credentials` and name `credentials`. This method returns `true` if an `ProtoUser` entity with matching username and password exists, which must have `true` as value of its `activated` property.

B.3. `protoRegister`

Class	`ProtoRegisterBean`
Business Interface	`ProtoRegister`
Scope	Conversation
Associated View	`proto-register.xhtml`
Description	Register a new user by creating a `ProtoUser` entity.

Methods

`void register()`	Register a new user by creating a `ProtoUser` entity. A username with the same name must not exist yet and the values of the `password` and `confirmPassword` properties of this component must be equal. The entity's `username` and `email` fields are set to the values of this component's corresponding properties. The password is encrypted using 256 iterations of a SHA1-HMAC-based hash with the randomly generated salt stored separately as another field of the user's entity. Newly registered users have their `activated` property set to `false` and must therefore confirm their account through the `protoActivator` property before login. If the registration succeeded, the `registrationSucceeded` property of this component is set to `true` and the `confirmationMail.xhtml` view is rendered internally, which sends out a mail containing the user's activation code to their address. Otherwise, `registrationSucceeded` remains `false` and status messages are added, detailing why the registration failed.
`void changePassword()`	Change password of currently logged in user. The old password must be entered into `oldPassword`. If it matches, the user entity's password hash is then set to the salted hash of the cleartext entered into `password` (`confirmPassword` must match).

Properties

`boolean registrationSucceeded` *(read-only)*	Whether the registration succeeded.
`String username`	Username for the new user.
`String email`	Email address for the new user.
`String password`	The new user's password in plaintext, which will be stored encrypted in the user's `ProtoUser` entity.
`String confirmPassword`	The new user's plaintext password again, to check whether it has been entered correctly.
`String oldPassword`	The old user's password, used when changing the password.
`String confirmationCode` *(read-only)*	The user's generated confirmation code, for inclusion in the confirmation mail.

B.4. `protoTranscript`

Class	`ProtoTranscriptImportBean`
Business Interface	`ProtoTranscriptImport`
Scope	Stateless
Associated View	`transcripts.xhtml`
Description	Import a student's transcript XML file.

See section 7.2 for details on the transcript XML file and how it is handled.

Internally, this component transforms the uploaded transcript XML into the more manageable XML scheme mentioned in 7.2 via XSLT and unmarshals the resulting XML's content into an object tree via JAXB. The unmarshalled objects, which contain the lectures that the student attended as well as the user's grades, are matched against existing `Lecturer` and `Lecture` entities. If no match exists for either of both entities, new instances are created with the information given in the objects. A new `Transcript` entity is created, as well as associated `Attendance` entities for each processed attendance.

Methods

`void importTranscript()`	Import a transcript. The transcript is read in from the input stream previously set as the `input` property. File upload, e.g. those of standard Facelets and RichFaces, can bind this property. Errors or warnings during any of the transcript import phases are added as status messages with appropriate severity.

Properties

`InputStream input`	The input stream from which the transcript XML is read.

C. Views

C.1. `about.xhtml`

URL Path	`/about`
Menu Entry	About
Description	A static “About” page containing acknowledgments.

C.2. `browseLectures.xhtml`

URL Path	`/browse`
Menu Entry	Rate
Associated Component	`lectureManager`
Action (before rendering)	`lectureManager.browseAllLectures`
Description	The lecture browser, which also allows students to rate lectures.

This view’s central element is the lecture table. The table uses pagination to show up to 200 lectures at once per page, and the pages can be chosen by a scroller immediately above the table. Further above the table are the filter controls, which allow users to filter according to lecture and lecturer name, semester in which lectures took place and, if logged in as a student, rated or unrated lectures.

The table iterates over a data model (implemented by the `LectureDataModel` class) given by the `lectureManager.rows` property. The iteration variable is of type `LectureBrowseRow`, a special class that contains properties for the associated `Lecture` entity, the `Attendance` entity if any and the number of total ratings given to the lecture so far.

Upon clicking any of the sort order links in the table’s headers, the `lectureManager.order` property is set to the order value corresponding to the chosen column and the `lectureManager.filter` action is invoked.

Unrated but attended lectures can be rated with the 5-star-input in the table’s “My Rating” column. Pressing any star sets the `lectureManager.pointsToSet` property and invokes the `lectureManager.rate` action. If an attended lecture has already been rated, the star input is replaced by a static display of the stars with the previously given rating (taken from `current.attendance.rating`, `current` being the iteration variable), together with an X-shaped control that removes the rating when clicked. Removing is achieved by invoking the `lectureManager.removeRating` action. For both the `rate` and `removeRating` actions, the lecture to operate on is defined by the data model’s current row.

Two of the filters, specifically those for filtering lecture and lecturer names, are simple text input boxes which are bound to `filterLecture` and `filterLecturer` in `lectureManager` respectively. They have simple submit buttons which invoke the `lectureManager.filter` action.

The semester filter is a drop-down list populated by the `lectureManager.filterSemesterItems` property, which is a list of `SelectItems`. An additional “All” item with an empty value is present in the list to not filter the table by semester. Choosing any value of the list box also invokes the `lectureManager.filter` action.

If logged in as a student, there are three additional filter links available: “All”, “Unrated” and “Rated”. Any time, only two out of the three links can be active, the inactive link denotes the current filter. Clicking the links invoke the `showAll`, `showOnlyUnrated` or `showOnlyRated` actions of the `lectureManager` component respectively. Those actions also invoke the `filter` action implicitly.

C.3. `editProfile.xhtml`

URL Path	`/editProfile`
Restrictions	Students only
Menu Entry	Edit Profile
Associated Component	`profileManager` / `editProfile`
Description	A panel for editing the student profile.

This view currently only contains radio buttons for choosing one’s gender (which may also be left as or set to “Unknown” for students that do not want to disclose their gender). An additional *Nickname* field is present and implemented, but hidden as the nickname is currently unused throughout the application.

The `editProfile` component only provides the action that is invoked when saving the profile, `editProfile.saveProfile`. All other inputs modify the properties of the current student entity, `profileManager.currentStudent`, directly.

C.4. `error.xhtml`

URL Path	*no explicit path, rendered on error*
Description	A page displaying error messages.

The generic error page is rendered whenever an uncatched exception occurred. Its displayed error messages are populated according to the type of the exception, as specified in `pages.xml`.

C.5. `feedback.xhtml`

URL Path	`/feedback`
Restrictions	Students only
Menu Entry	Feedback
Associated Component	`feedback`
Description	A page with a feedback form.

Logged in students can provide direct feedback to a fixed email address using this form. Its textarea is bound to the `feedback.feedback` property, and submitting invokes the `feedback.send` action. See the description of the `feedback` component in section 6.3 for details.

C.6. `home.xhtml`

URL Path	`/home`
Menu Entry	Home
Description	A mostly static welcome page.

This welcome page statically provides a welcome message with links for browsing lectures and logging in when no student is logged in, and an explanation of the steps to use `eval` when a student is logged in. If the student has been ranked in the past, it also displays the student's last rank and the total number of students that were ranked on the last run.

C.7. `login.xhtml`

URL Path	`/login.seam` and implicitly when trying to access a restricted page
Menu Entry	Login
Description	The page through which users login.

This page allows a user to login. The properties for username (`credentials.username`) and password (`credentials.password`), as well as the login action `identity.login` are provided by Seam Security.

C.8. `projects.xhtml`

URL Path	`/projects`
Menu Entry	Projects
Description	A static page containing available student projects.

C.9. `showRanking.xhtml`

URL Path	`/ranking`
URL Parameters	`ranking`: mapped to `showRanking.selectedRankingId` *(not required)*
Menu Entry	Ranking
Associated Component	`showRanking`
Action (before rendering)	`showRanking.showRanking`
Description	The ranking browser.

The top part of the page shows the selected ranking taken from the `showRanking.displayRanking` property as a single table (if not *null*). The table iterates through `showRanking.displayRanking.ranks` which is a list of `Rank` entities.

The bottom part of the page shows all available rankings, from the `showRanking.rankings` property. All rankings, except for the currently displayed ranking, are represented as clickable links which set the `showRanking.selectedRankingId` property through the `ranking` URL parameter.

For logged in Administrators (users having the `admin` role), an additional "generate!" button is displayed that invokes the `showRanking.generateRankingNow` action.

C.10. `transcripts.xhtml` (non-prototype part)

URL Path	`/transcripts`
Restrictions	Students only
Menu Entry	Transcripts
Associated Component	`transcriptManager`
Description	The transcript browser and (prototype) upload area.

For a description of the prototype transcript import portion of this page, see section 7.4.

The rest of the page shows already uploaded transcripts as a list, which iterates through the `transcripts` property (consisting of `Transcript` entities).

For each transcript, a table iterating through `attendances` of the current transcript is shown. This table details the `Attendance` entities belonging to this transcript.

Each transcript also has a "Revoke" button, which invokes the `transcriptManager.revoke` action.

D. Prototype Views

D.1. `transcripts.xhtml` (prototype specific part)

URL Path	`/transcripts`
Restrictions	Students only
Menu Entry	Transcripts
Associated Component	`protoTranscript`
Description	The transcript browser and (prototype) upload area.

The top part of this view is specific to the prototype and allows users to import transcript XML files into the system. An `<s:fileUpload>` control is bound to the `protoTranscript.input` property (as an `InputStream`) and submitting invokes `protoTranscript.importUpload`.

D.2. `proto-activate.xhtml`

URL Path	`/proto-activate`
URL Parameters	`username`: mapped to `protoActivator.username` *(required)*, `code` *(required)*: mapped to `protoActivator.activationCode` *(required)*
Menu Entry	*none* (reached by direct link)
Associated Component	`protoActivator`
Action (before rendering)	`protoActivator.activate`
Description	Confirmation page for activation.

Upon visiting this view, the two URL parameters `username` and `code` set `protoActivator`'s `username` and `activationCode` property respectively. Subsequently, `protoActivator`'s `activate` action is invoked.

If this resulted in the successful activation of a user, a short, static confirmation message is displayed with a link to the Web Application's login page. Otherwise, an equally static error message is displayed.

This view is meant to be reached by a link in the confirmation mail a new user received after registering.

D.3. `proto-changePassword.xhtml`

URL Path	`/proto-changePassword`
Restrictions	Logged in users only
Menu Entry	Password
Associated Component	`protoRegister`
Description	Change user password.

Consists of a form that populates the `oldPassword`, `password` and `confirmPassword` properties of the `protoRegister` component and invokes `protoRegister.changePassword` when submitted.

D.4. `proto-confirmationMail.xhtml`

URL Path	`/proto-confirmationSent`
Menu Entry	*none* (reached by navigation rule from `proto-register.xhtml`)
Associated Component	`protoRegister`
Description	Confirmation page displayed after initial registration.

If registration succeeded, this page is displayed. It contains a mostly static text that thanks the user for registering and describes how to activate the new account.

D.5. `proto-register.xhtml`

URL Path	`/proto-register`
Menu Entry	Register
Associated Component	`protoRegister`
Description	User registration form.

Consists of a form that populates the properties of the `protoRegister` component and invokes `protoRegister.register` when submitted.

If registration was successful, the user is sent to the |proto-confirmationSent.xhtml| view by the following navigation rule in pages.xml:

```
<navigation>
<rule if="#{protoRegister.registrationSucceeded}">
<redirect view-id="/proto-confirmationSent.xhtml" />
</rule>
</navigation>
```

Otherwise, error messages detail why registration failed as part of the form.

Bibliography

[1] M. Aktas, M. Nacar, and F. Menczer. Using hyperlink features to personalize web search. *Advances in Web Mining and Web Usage Analysis*, pages 104–115, 2006.

[2] C. Bauckhage. Image tagging using pagerank over bipartite graphs. *Pattern Recognition*, pages 426–435, 2008.

[3] Hans Bergsten. *JavaServer Faces*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. ISBN 0596005393.

[4] M. Bianchini, M. Gori, and F. Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[5] N.W.H. Blaikie. *Analyzing quantitative data: From description to explanation*. Sage, 2003.

[6] P. Bonacich. Power and centrality: A family of measures. *American journal of sociology*, pages 1170–1182, 1987.

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, April 1998. ISSN 01697552. doi: 10.1016/S0169-7552(98)00110-X. URL `http://dx.doi.org/10.1016/S0169-7552(98)00110-X`.

[8] François Bry, Norbert Eisinger, Tim Furche, Christoph Wieser, and Harald Zauner. Web-informationssysteme, 2011. Lecture in the winter semester of 2011/2012 at Ludwig-Maximilians-Universität München, `http://www.pms.ifi.lmu.de/lehre/webinfosys/11ws12/`.

[9] K. Bryan and T. Leise. The $25,000,000,000 eigenvector. *The linear algebra behind Google*, 2005.

[10] B. Burke and R. Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly Media, 2006.

[11] L. Cabral and A. Hortacsu. The dynamics of seller reputation: Theory and evidence from ebay. Technical report, National Bureau of Economic Research, 2004.

[12] Ying Ding, Erjia Yan, Arthur Frazho, and James Caverlee. PageRank for ranking authors in co-citation networks. *Journal of the American Society for Information Science and Technology*, 60(11):2229–2243, 2009. ISSN 1532-2890. doi: 10.1002/asi.21171. URL `http://dx.doi.org/10.1002/asi.21171`.

[13] Y. Du, Y. Shi, and X. Zhao. Using spam farm to boost pagerank. In *Proceedings of the 3rd international workshop on Adversarial information retrieval on the web*, pages 29–36. ACM, 2007.

[14] N. Eiron, K.S. McCurley, and J.A. Tomlin. Ranking the web frontier. In *Proceedings of the 13th international conference on World Wide Web*, pages 309–318. ACM, 2004.

[15] C. Gerlitz and A. Helmond. Hit, link, like and share. organizing the social and the fabric of the web in a like economy. In *Paper presented at the DMI mini-conference*, volume 24, page 25, 2011.

[16] Z. Gyongyi and H. Garcia-Molina. Web spam taxonomy. In *First international workshop on adversarial information retrieval on the web (AIRWeb 2005)*, 2005.

[17] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, and G. Golub. Computing pagerank using power extrapolation. *Stanford University Technical Report*, 2003.

[18] T.H. Haveliwala and S.D. Kamvar. The second eigenvalue of the google matrix. Technical report, Citeseer, 2003.

[19] R. Hightower. Facelets fits jsf like a glove. *IBM Developer Works*, 2006.

[20] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Folkrank: A ranking algorithm for folksonomies. *Proc. FGIR*, 2006, 2006.

[21] S. Jamieson. Likert scales: how to (ab) use them. *Medical Education*, 38(12):1217–1218, 2004.

[22] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1): 39–43, 1953.

[23] W. Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 26(2):120–125, 1983.

[24] G.E. Krasner and S.T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[25] A.N. Langville and C.D. Meyer. Deeper inside pagerank. *Internet Mathematics*, 1(3): 335–380, 2004.

[26] Stephan Link. Eigenvalue-based bipartite ranking. Bachelor Thesis, 2011.

[27] M.E.J. Newman. *Networks: an introduction*. Oxford Univ Pr, 2010.

[28] C. Richardson. What is pojo programming? *Java Developer's Journal, http://java.sys-con.com/node/180374*, 2006.

[29] Daniel A. Spielman. Graphs and networks lecture 16 pagerank and random walks on directed graphs, 2010.

[30] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Relevance search and anomaly detection in bipartite graphs. *ACM SIGKDD Explorations Newsletter*, 7(2):48–55, 2005.

[31] Y. Sun and C. Giles. Popularity weighted ranking for academic digital libraries. *Advances in Information Retrieval*, pages 605–612, 2007.

[32] P.S. Yu, X. Li, and B. Liu. On the temporal dimension of search. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 448–449. ACM, 2004.

[33] M.J. Yuan and T. Heute. *JBoss seam: simplicity and power beyond Java EE*, volume 1. Prentice Hall PTR, 2007.