Seminarkurs Schulbuchverwaltung

Schüler:
Kay Stipcevic,Fabius Balogh, Jan Lafferton, Yarin Gora

Lehrer:
Herr Würz

Inhalt

Präambel

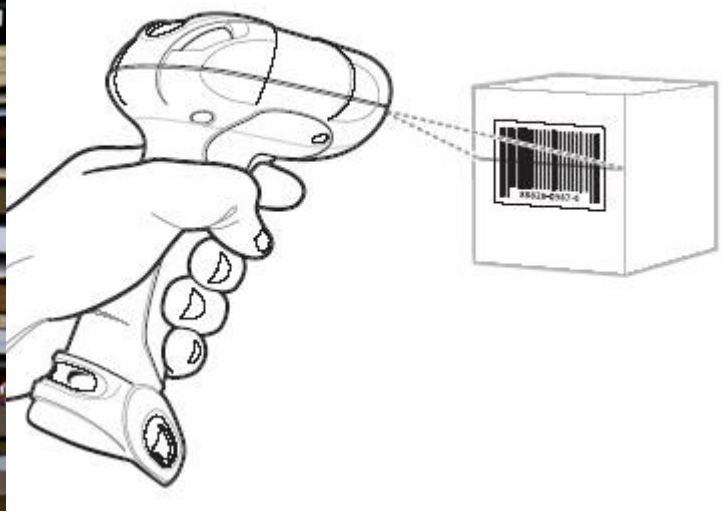
Eigenständigkeitserklärung

Hiermit bestätigen wir, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt haben. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle im Quellenverzeichnis kenntlich gemacht.

Einleitung - Was ist die Schulbuchverwaltung?

Die Schulbuchverwaltung der AES-Ettlingen ist ein Programm, welches dazu dient, die Bücher, die Schüler zur Unterstützung des Unterrichtsstoffs erhalten zu verwalten.

Dazu werden alle Schüler, Bücher und deren Kopien in einer Datenbank gespeichert, welche von einer graphischen Oberfläche aus veränder- und einsehbar sind. Alle Daten sind einzeln oder als ganzes abrufbar. Des weiteren kann man das Einsammeln und Aushändigen von Büchern per Barcode durchführen, mit dem alle Bücher versehen werden. Die Generierung der Barcodes übernimmt das Programm ebenfalls. Ein Barcodescanner kann diese dann lesen und im richtigen Reiter des Programms automatisch erkennen, um welches Buch es sich handelt, sowie zu welchem Schüler es gehört.



Dieses Programm erhält seit mehreren Jahren von Seminarkursen diverse Softwarelösungen, welche bisher aber immer einige Probleme aufwiesen und nun wieder eine weitere Gruppe benötigten.

Einleitung - Zielsetzung für den Seminarkurs

Zu Beginn des Jahres trafen wir uns um die Rahmenbedinungen für den Seminarkurs zusammen mit Herrn Würz festzulegen. Die folgende Liste resultierte aus dem Gespräch:

- User Login
  - User Einloggen(Admin,Lehrer,Schüler)
- Schüler
  - Schülern zugeordnete Bücher ausgeben
    - Schüler Versetzen in verschiedene Klassen
    - Anzahl Bücher
    - Titel
    - Schüler Name
    - Klasse
    - Ausgeteilt
    - Gekauft
    - Bezahlt
      - Bezahl boolean
      - alle noch zu bezahlen den mit Haken
      - alle auf einmal abhaken
    - ISBN
    - Drucken Listen PDF-Export
- Bücher
  - Übersicht Über Alle Bücher
    - Bücher hinzufügen/entfernen
    - Für einen Schüler
    - Für eine komplette Klasse
  - Neue Bücher in das System aufnehmen
    - in die Datenbank / aus der DB entfernen
  - Barcode
    - einscannen
    - drucken
  - Identifizieren Buchhalter
  - Drucken Listen PDF-Export
- Klassen
  - Klassen anlegen / entfernen
  - Schüler hinzufügen / entfernen
  - Einsammeln
  - CSV Dateien einlesen
  - Namensänderung
  - Drucken von Listen, PDF-Export
- Datenbank
  - Backup (Datei)
  - Löschfunktion

Einleitung - Probleme, die behoben werden mussten

Der alte Seminarkurs hatte diverse Probleme mit Leistung und Bedienung der Oberfläche, welche behoben werden mussten. Beim Wechseln auf andere Reiter lud das Programm teilweise für zweistellige Sekundenzahlen, was das alltägliche Arbeiten mit der Software frustrierend ausfallen lies. Außerdem musste unnötig oft der Reiter gewechselt werden. Diese Kombination stellte sich als äußerst ungünstig heraus, und bedurfte eines neuen Seminarkurses um behoben zu werden.

Nach Analysieren des (unkommentierten und unformatierten) Quellcodes der vorherigen Gruppe traten einige fundamentale Probleme in den Vordergrund, die gelöst werden mussten um der Situation Herr zu werden:

- Zunächst war der Code durch und durch unstrukturiert und offensichtlich nicht für die Weiterarbeit geplant.
- Das Programm war außerdem weitgehend in funktionaler und redundanter Architektur gehalten, was die Stärken der gewählten Programmiersprache, Java, größtenteils negierte.
- Die Datenbankabfragen waren ebenso redundant gestaltet und benötigten teilweise absurd viele Prozesse um Ergebnisse zu liefern. Dies war der Hauptgrund für die Performanzprobleme.
- Die Architektur der Datenbank war umständlich und bot nur sehr ineffiziente Zugriffe.

Diese Komplikationen führten dazu, dass wir uns entschieden das Projekt komplett neu aufzusetzen, um den Planungsfehlern ein Ende zu setzen.

In der Planungsphase galten folgende Fragen zu klären :

1. Für welche Programmiersprache wir uns entscheiden sollten
2. Ob das Programm serverseitig oder nativ laufen soll
3. Wie die neue Datenbankarchitektur aussehen wird

Auswahl des „TechStack“

Der „Techstack“ beschreibt die im Projekt verwendete Technologie, sprich Programmiersprache, Frameworks, Server, usw.

Wir entschieden uns für eine serverseitige Applikation, da die Datenbank um einiges sicherer ist, wenn sie auf einem zentralen Server gespeichert wird, anstatt auf einem einzelnen Endgerät. Außerdem öffnete dies uns die Tore für „Ruby on Rails“, einem auf der Programmiersprache „Ruby“ basierendem Framework, welches durch seine vielen Möglichkeiten zur Generierung von Code, extrem schnell ist, wenn des darum geht Ergebnisse zu erzielen. Daher auch der Name „on Rails“ oder „auf Schienen“ im Deutschen. Es ist eine extrem effiziente Möglichkeit das Ziel zu erreichen, wenn man die vorgefertigten Wege, über den generierten Code nimmt.
Außerdem bietet Ruby das „RubyGems“ Paketsystem(https://rubygems.org/), welches neue Pakete, Frameworks oder generelle Hilfen in einem schnellen Konsolenbefehl nahtlos ins Projekt integriert. Dieses System stellte sich mehrmals als Retter in Not heraus, denn unsere initialen Probleme mit der Generierung von PDF-Dateien konnten wir einfach mit dem „Prawn“-Gem lösen, welches das Fundament für solch einen Vorgang setzte. Daher kommt auch eines der Paradigmen von Ruby und Ruby on Rails „If it has been done once, you don't have to do it again” ,zu Deutsch „Wenn es schon einmal gemacht wurde, musst du es nicht noch einmal machen.“.
Unsere Datenbank gestalteten wir in Sqlite3, einer Alternative zum vorherig verwendeten MySQL, die einfachere Zugriffe und mehr Mobilität bietet.

Illustration 1: Ruby on Rails Logo

Unterschiede zum bisherigen Schulstoff

Der IT-Unterricht im Rahmen unseres Proflifachs Informationstechnik hat den Bedarf an Fähigkeiten für unser Projekt nur teilweise decken können. Zum Einen lernten wir bisher nur in den Sprachen „Java“ , „Assembler“ und „C“ das Programmieren, zum Anderen haben wir in keiner der genannten die Webentwicklung angesprochen. Der Punkt, welcher im Ansatz gedeckt war, waren Datenbanken. Im Software-Unterricht haben wir mit SQL gelernt, Datenbankabfragen zu gestalten. Dies stellte sich als hilfreiche Grundlage heraus, allerdings auch nicht mehr als das. Ein Fundament. Das Designen einer effizienten Datenbank, sowie die diversen Probleme die auftreten können, wenn dies nicht durchgeführt wird, war uns zu Beginn unbekannt.

Da wir uns für „Ruby on Rails“ entschieden, mussten wir ebenfalls eine neue Programmiersprache erlernen, welche sich von den in der Schule behandelten stark unterscheidet.

Ein letzter Punkt ist die Arbeitsweise. Als Schüler waren wir eine strengere Regulierung der Arbeitsphasen gewohnt und fanden uns oftmals kurz vor einem Termin ohne Ergebnisse, da wir zu lange zögerten, die besprochenen Punkte anzugehen.

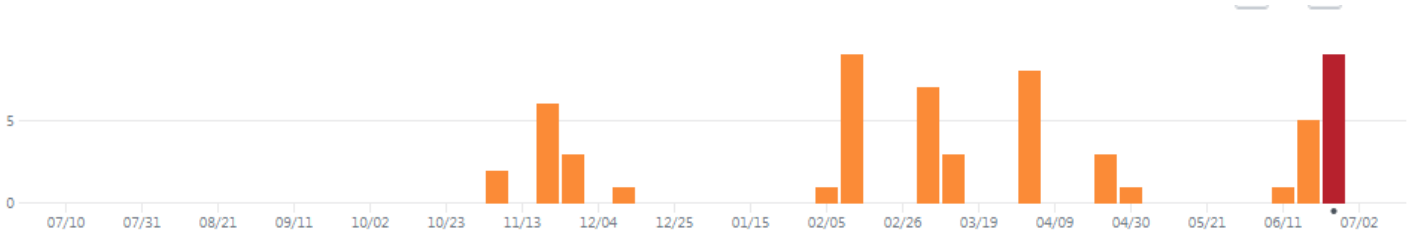


Illustration 2: Visualisierung unserer Github-Commits, in Abhängigkeit vom Datum

All diese Punkte in Betracht ziehend, ist unsere relativ lange Planungsphase zu Beginn des Projekts, auch in retrospektive betrachtet, sinnvoll, denn das Arbeiten im Seminarkurs war drastisch anders, als das im Unterricht. Als Schüler im Profilfach Informationstechnik ließen wir uns aber dennoch nicht unterkriegen, teils aus Interesse am Fach, teils aus Spaß am freien Arbeiten.

Auswahl des Workspace

Nach Recherchen hatten wir uns auf zwei mögliche Entwicklungsumgebungen verständigt. Die erste war **Atom** (https://atom.io), allerdings tauchten Probleme beim Aufsetzen eines lokalen Servers unter Windows auf. Daher stiegen wir auf unsere derzeitige Entwicklungsumgebung **Cloud 9** (https://c9.io) um. Der entscheidende Vorteil dieser Plattform ist, dass man nicht nur Echtzeitzugriff übers Web hat, damit also überall und gleichzeitig am Projekt arbeiten kann, sondern auch dass Cloud9 dem Nutzer einen Ubuntu Server zur Verfügung stellt, auf welchem Ruby on Rails ohne weitere Umwege funktioniert.

Datenbankplanung

Die Datenbank wurde neu von Grund auf neu strukturiert, da die alte Architektur zu massiven Leistungsdefiziten führte.

Wie In der Abbildung zu sehen, sind nur vier Tabellen vorhanden. Die alte Architektur hatte weitaus mehr und war so verschachtelt aufgebaut, dass die Access Paths, die Wege die das Programm geht um die Daten aus der Datenbank zu erhalten, unnötig lang waren. Des weiteren entfernten wir einige Redundanzen indem wir, anstatt eine „Copieshistory“ zu haben, in welcher alle archivierten oder noch zu bezahlenden Kopien waren, eine „Archiv“-Klasse einführten, in der Schüler gespeichert werden, welche die Schule nicht mehr besuchen, aber dennoch zu bezahlende Kopien haben. Kopien, die von einem Schuljahr ins nächste noch ausstehen, werden einfach auf dem jeweiligen Schüler gespeichert, denn in der „Kopie“ Tabelle, wird direkt der Schüler referenziert.

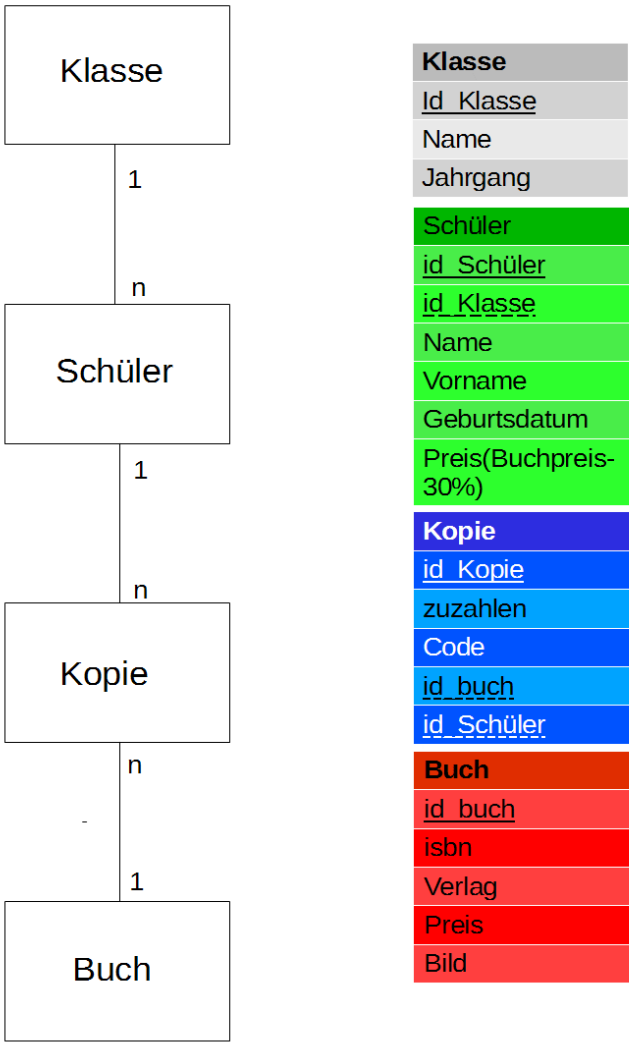


Illustration 3: Architektur der aktuellen Datenbank

Einrichten eines Github-Projekts

Um nicht nur Backups für unsere Arbeit zu schaffen, sondern auch die Gruppenorganisation und Zielsetzung zu vereinfachen, entschlossen wir uns ein Github Projekt für den Seminarkurs anzulegen. Dieses Projekt ist nach wie vor aufzufinden unter
https://github.com/Hummeltron/SBV-AES

und beinhaltet die Historie unserer Arbeit, sowie unserer Projekt per-sé, welches zum Download bereitsteht.

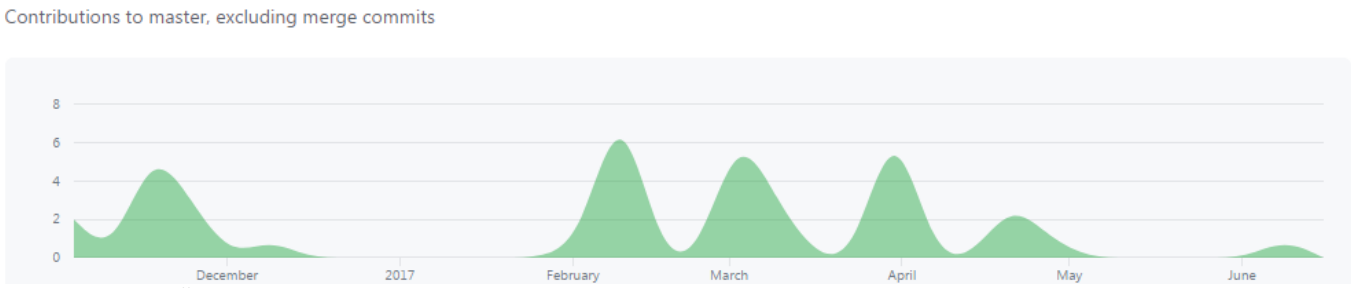


Illustration 4: Änderungshistorie im Projekt

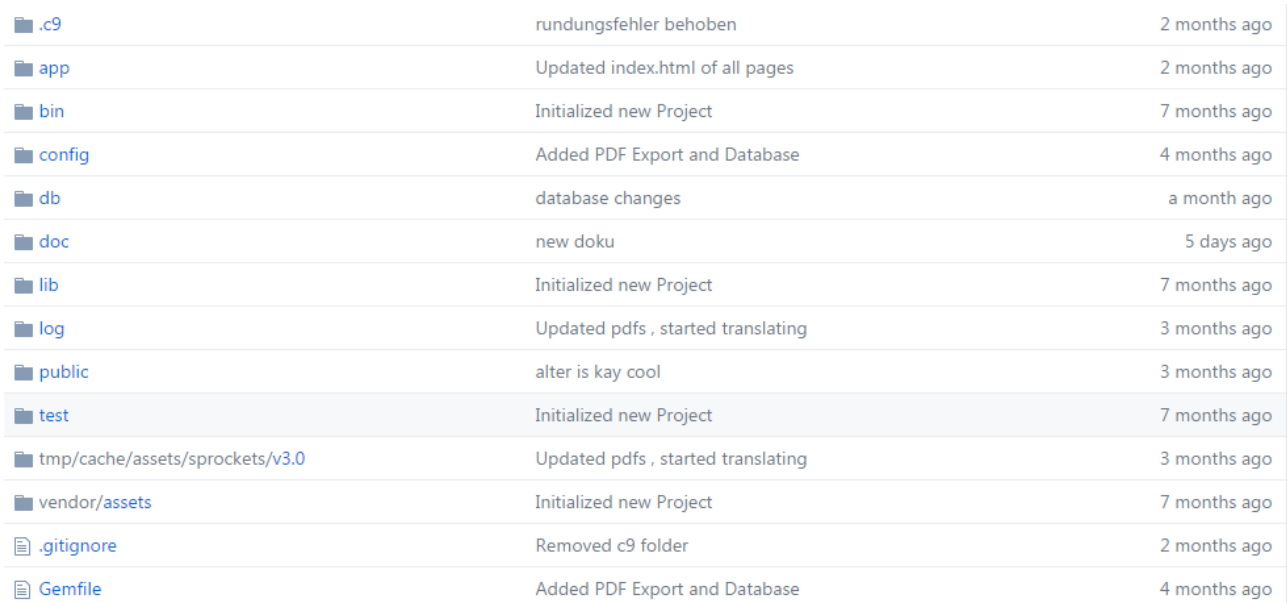


Illustration 5: Projektanzeige auf der Github-Weboberfläche

Das Architekturmuster

Für unser Projekt wählten wir das “Go-to” der Webentwicklung, das MVC oder “Model-View-Controller”-Muster. Wie der Name schon schließen lässt, ist ein MVC-Programm in drei Sektionen unterteilt, Model, View und Controller.

Der View ist, was der User im Browser sieht. Hier werden die HTML Elemente des Programms zusammengefügt, um in Korrelation mit den anderen beiden Sektionen das Bild in den Browser zu bringen.

Der Controller hält die Logik des Programms. Operationen vom View werden an den Controller weitergegeben um verarbeitet zu werden. Beispielsweise ein Klick auf einen Button an der Oberfläche wird im Controller verarbeitet und mit dem Nötigen Algorithmus versehen, um letztendlich zum Ergebnis zu führen.

Das Model verwaltet die Datenbank. Vom Controller aus kommen Änderungen an, welche der User im View “beantragt” hat. Der Controller interagiert dann mit dem Model, welches die Datenbankarchitektur einsehen kann, um die nötigen Änderungen geordnet durchzuführen, und Konsistenz beizubehalten. Man kann also sagen, dass die Verwaltung der Daten dem Model zufällt.

Ein großer Vorteil dieser Architektur ist die Sicherheit. Durch die strikte Unterteilung lässt sich Frontend und Backend, also user- und serverseitiger Zugriff unterteilen und der User kann damit nicht in Daten einsehen, welche ihm nicht zustehen. Der Controller und das Model sind meist serverseitig, während der View beim User angesiedelt ist, damit er die Oberfläche bedienen kann. Ein möglicher Nachteil ist der mit dem Muster verbundene Aufwand, die Applikation so strikt zu unterteilen, allerdings ist dies in unserem Falle durch die Größe des Projekts und die Generatoren von Ruby und Rails nicht relevant.

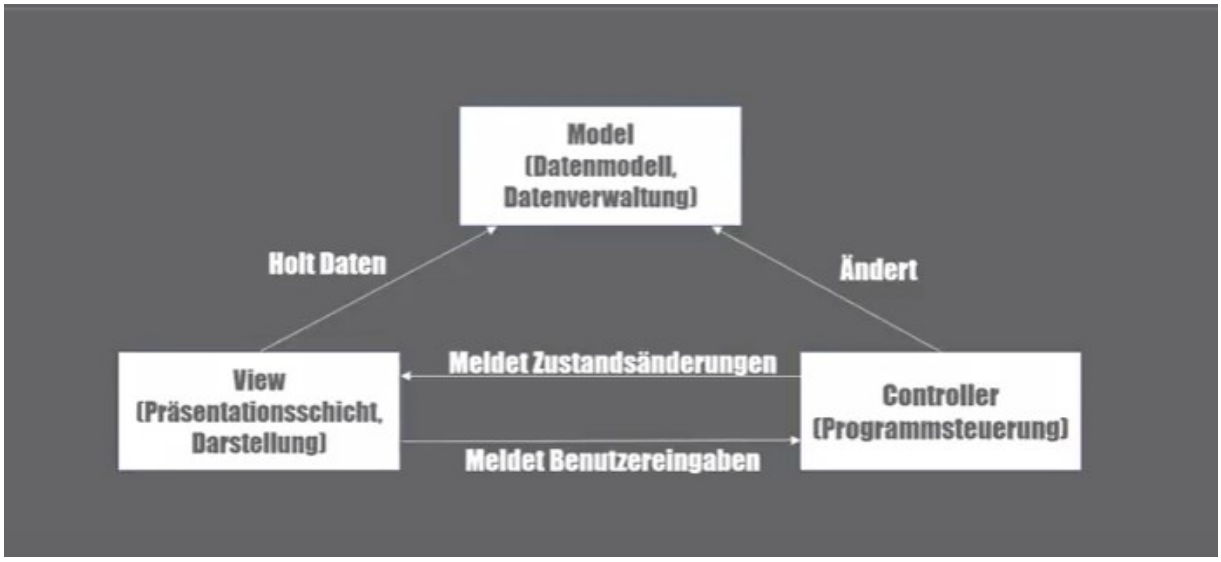


Illustration 6: Diagramm der "MVC"-Architektur

Erstaufsetzen des Projekts

Wie schon mehrmals erwähnt, profitiert Ruby on Rails von einer Vielzahl an „Quality of Life“-Tools, unter anderem auch, einer automatischen Funktion zum Anlegen von neuen Projekten.

Mit dem Befehl `$Rails new <PROJEKTNAME>` kann durch die Kommandozeile im aktuellen Ordner eine Grundstruktur erstellt werden, in der Ruby on Rails weiterhin problemfrei das MVC-Muster generieren kann.

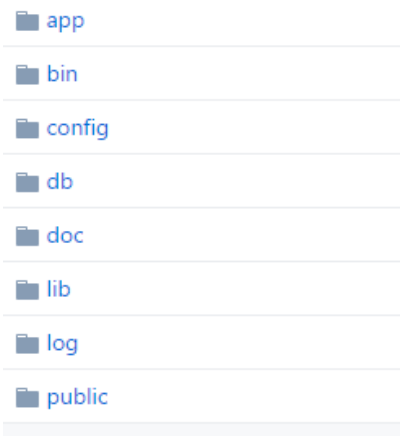


Illustration 7: Von Ruby on Rails generierte Ordner

Diese Struktur, welche in allen Ruby on Rails Projekten zu finden ist, wurde nun erstellt. Im „app“-Ordner befinden sich die jeweiligen Controller und Views, sowie alle Scripts. Das Model ist im „db“-Ordner, kurz für „Database“ oder Datenbank, zu finden. Die Ordner „bin“, „config“, „lib“ und „log“ enthalten fundamentales, wie zum Beispiel die „rake“-Datei, welche für diverse Datenbankoperationen angesprochen wird. Im „Public“-Ordner können jegliche noch anstehende zusätzliche Daten gespeichert werden. Diese sind zum Beispiel die Bilder, welche wir für die Bücher hinzufügten. Zuletzt ist noch „doc“ zu sehen. Dies war lediglich ein Ordner den wir selbst anlegten um die Dokumentation und das Protokoll zu Speichern.

Alle Gems von Ruby werden außerdem noch in der „Gemfile“ gespeichert, die schlicht im Projekt ohne ein weiteres Verzeichnis liegt. Sie wird vom Programm verwendet um Versionen zu vergleichen und Updates durchzuführen.

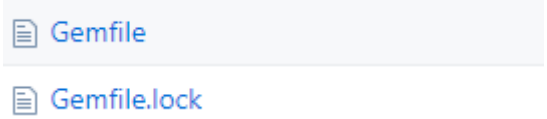


Illustration 8: In der Gemfile sind alle verwendeten Gems referenzeiert

„MVC“ konfiguriert in Ruby on Rails

Um eine Datenbank in Ruby on Rails aufzusetzen, muss man zunächst selbstredend in der Kommandozeile in den Projektordner navigieren. Dies kann mit folgendem Beispielbefehl durchgeführt werden :

```
$ cd C:\Users\test\Documents\SBV-AES
```

„cd“ steht für „Change Directory“, zu Deutsch „Ordner wechseln“.

Nun gibt es die Methode „scaffolding“ in Rails, welche automatisch Model, View und Controller mit beliebigen Parametern erstellt. „Scaffold“ bedeutet auf Deutsch treffend „Gerüst“, denn dies ist der Zweck der Methode. Sie erstellt die Basis für die Website im Model-View-Controller Schema. Der Befehl sieht wie folgt aus :

```
$ rails generate scaffold <TABLE> <COLUMN:DATATYPE> , <COLUMN:DATATYPE>
```

Da Controller und View zunächst eine sehr statische Interaktion haben (hauptsächlich Operationen wie das Befüllen von Tabellen), ist es hier nur von Nöten die Tabelle in der Datenbank anzugeben, die man dazu erstellen möchte.

Nun ist neben View und Controller die Tabelle erzeugt und liegt im Model. Dieses muss nun lediglich in die Datenbank migriert werden, damit das neue Schema übernommen wird. Auch hier kann Ruby on Rails dem Entwickler viel Arbeit ersparen, indem der Befehl

```
$ rake db:migrate
```

exakt das beschriebene erledigt. Das „Schema“ eine Datei, die die Tabellen und ihre Spalten speichert wird hier auf die Datenbank angewandt, damit die Änderungen übernommen werden. Diese „Schema“-Datei ist wie folgt strukturiert :

```
create_table "books", force: :cascade do |t|
  t.string   "label"
  t.float    "price"
  t.string   "isbn"
  t.string   "image"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.boolean  "topay"
end
```

Illustration 9: Tabelle "books" in der Schema-Datei

Die einzelnen Spalten stehen hier mit ihrem Datentyp versehen untereinander.

Datenmigration in die neue Datenbank

Eine der Aufgaben war es, die Daten aus der alten Datenbankarchitektur in unsere neue zu migrieren. Dies erfolgte durch einige gezielte SQLite-Queries, Abfragen der Datenbank, welche die Datensätze verändern können.

Zunächst war es wichtig, die alten Daten in eine Datenbank mit den neuen zu bringen. Hierfür erstellten wir in der neuen DB einfach die nötigen Tabellen und verschoben alle Daten in diese. Nun konnten wir ähnliche Spalten direkt kopieren. Zum Beispiel die Namen der Schüler, deren Geburtsdatum und ihre Klasse konnten wir direkt übernehmen. Doch nicht bei allen Tupeln lief es so reibungslos. Kleinere Anpassungen mussten beispielsweise bei den Kopien gemacht werden. Die alte Architektur hat die Kopien in zwei größtenteils redundanten Tabellen gehalten, den „Copies“ und der „Copieshistory“. In den „Copies“ wurden die Kopien per-sé gespeichert, welche mit einem Code versehen waren. In der „Copieshistory“ waren zu diesen genannten Kopien die Schüler gespeichert. Wir lösten dies in einer einzigen Tabelle und deswegen fielen einige Komplikationen an. Die Datensätze der beiden alten Tabellen hingen natürlich in Relation zueinander und mussten deshalb geordnet übernommen werden. Deshalb erstellten wir eine weitere temporäre Tabelle, die alle Spalten von „Copies“ und „Copieshistory“ enthielt. Erst setzten wir alle Daten der „Copies“ hinein indem wir sie schlicht kopierten. Dann führten wir eine Query durch, die die „Copieshistory“ so kopiert, dass alle Daten am richtigen Ort bei den referenzierten „Copies“ sind. Die Query lautete wie folgt :

```
UPDATE copies_copieshistory_temp

SET student_id = copieshistory.student_id

WHERE copieshistory.copieshistory_id = copieshistory_id
```

Dies übernahm die Werte an exakt den Stellen, an denen sich die einzigartigen Ids gleichten.

Anschießend konnten wir die angepassten Datensätze in unsere neue Tabelle übernehmen und hatten eines der großen Performance-Probleme, die zu langen Access Paths, gelöst. Die alte Datenbank wurde enorm ineffizient, denn bei jeder Abfrage mussten zunächst alle ungefähr 8000 „Copies“ mit gleich vielen Datensätzen in der „Copieshistory“ verglichen werden. Dies führte also bei jeder Abfrage dazu dass 8000 mal 8000 Datensätze überprüft wurden, was 64000 Einzelabfragen gleicht. Für diese 8000 Sätze haben wir in der neuen Architektur das logische Minimum, nämlich 8000 Abfragen. Damit haben wir die Leistung des Programms buchstäblich exponentiell gesteigert.

Routes

Die Routes dienen als Referenz für den View. Die Datei ist im „Config“-Ordner angesiedelt und „zeigt“ mit einer URL-Endung auf den jeweiligen View.

Im Bild unten ist beispielsweise die Zeile `get '/classes'  => 'aesclasses#index'` angegeben. Diese sagt aus, dass die „index“- Datei im View „aesclasses“ auf der Endung „/classes“ erreichbar ist. Im Browser würde es bei einer Website dann folgendermaßen aussehen : www.beispielwebsite.com/classes .

Ohne die Routes wüsste das Programm also nicht, welche Views es wann anzeigen sollte.

Im Bild über dem genannten Befehl ist auch die Zeile `resources :aesclasses` zu sehen. Diese gibt einfach die Referenz zum jeweiligen View an. Unten steht „aesclasses#index“. Die „aesclasses“ stammen aus der „resources“ Zeile. Sie zeigen also auf den View, welcher dann mit einer bestimmten URL-Endung aufgerufen werden kann. Die „Routes“-Datei ist damit essentiell für das Programm.

```
Rails.application.routes.draw do
  resources :bookdetails
  resources :books
  resources :copies
  resources :students
  resources :aesclasses
  # For details on the DSL available within this file, see http://guides.ruby

  get '/'             => 'aesclasses#index'
  get '/classes'      => 'aesclasses#index'
  get '/students'     => 'students#index'
  get '/books'        => 'books#index'
  get '/bookdetails'  => 'bookdetails#index'
  get '/copies'       => 'copies#index'
end
```

Illustration 10: Die Pfade werden in der Routes-Datei referenziert

Grundlegendes Design – HTML, CSS, Bootstrap, JS

Das Frontend unserer Website benötigte selbstverständlich ein intuitives Design, welches die UX, die User Experience, angenehmer gestaltete als die alte Oberfläche. Hierbei wählten wir natürlich das Go-To der Webentwicklung, „HTML“ und „CSS“.

Neben dem beliebten HTML, welches jeden View buchstäblich von Oben nach Unten anzeigt, dient CSS der Formatierung von diesem, durch sogenannte „Style-Sheets“. Mit diesen kann man die HTML-Elemente verankern und formatieren, sowie diverse On-Click-Funktionen ausführen lassen.

Auf CSS liegt wiederrum „Bootstrap“, ein Framework, welches diverse Design-Schemata zur Verfügung stellt. Bootstrap greift außerdem auf Javascript (auch JS genannt) zu, um die Designs „responsive“ werden zu lassen. Das bedeutet, dass diese sich dem jeweiligen Endgerät in Größe und anordnung anpassen, damit die Website niemals verschoben oder ungeordnet wirkt.

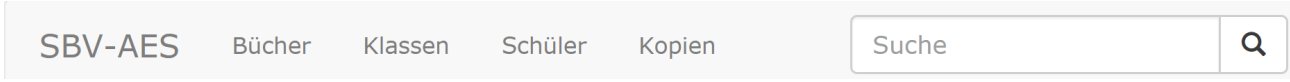


Illustration 11: Unsere Navigationsleiste nutzt die Bootstrap-Vorlage "Navbar"

```
15                  <ul class="nav navbar-nav">
16                      <li><%= link_to "Bücher", "/books" %></li>
17                      <li><%= link_to "Klassen", "/classnames" %></li>
18                      <li><%= link_to "Schüler", "/students" %></li>
19                      <li><%= link_to "Kopien", "/copies" %></li>
20                  </ul>
```

Illustration 12: Der Code zur "Navbar"

Im gezeigten Code wird lediglich die „Navbar“-Klasse angesprochen (Zeile 15), welche dann die jeweiligen Views, zu denen sie leiten soll referenziert (Zeile 16-19).

Die Korrelation von HTML, CSS und Bootstrap ist damit für unser Design fundamental, da sie uns diverse, modern aussehende Vorlagen bietet.

Verändern der Datensätze, ausgehend von der Oberfläche

Selbstverständlich ist es von der Oberfläche aus möglich, einzelne Einträge, wie zum Beispiel Schüler, zu bearbeiten. Dies geschieht für den User lediglich durch Befüllen einiger Felder und Klicken von Buttons. Auf der Softwareebene allerdings, ist dieser Vorgang ein wenig anders.

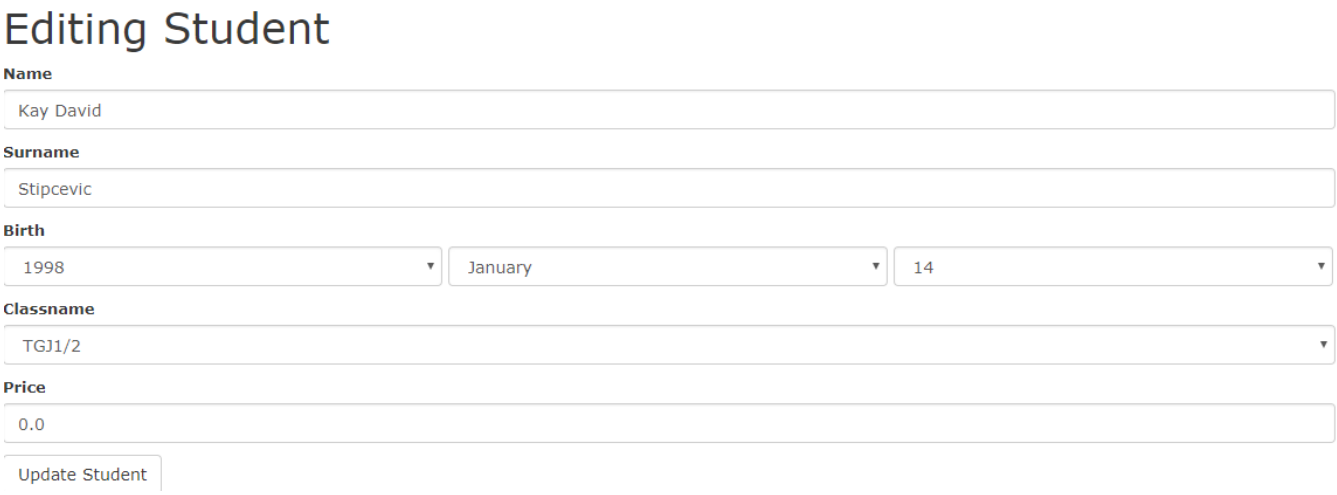


Illustration 13: Bearbeiten eines Schülers auf der Oberfläche

Beim Klicken auf den „Update“-Button wird die jeweilige Methode im Controller aufgerufen.

```
71      def update
72          respond_to do |format|
73              if @student.update(student_params)
74                  format.html { redirect_to @student, notice: 'Student was successfully updated.' }
75                  format.json { render :show, status: :ok, location: @student }
76              else
77                  format.html { render :edit }
78                  format.json { render json: @student.errors, status: :unprocessable_entity }
79              end
80          end
81      end
```

Illustration 14: Methode zum Verändern von Students

Diese nimmt die jeweiligen angegebenen Parameter (in Zeile 73 „Student_Params“) und gibt sie mit der „update“ Funktion weiter, um die Datensätze des jeweiligen Schülers zu verändern.

Dieser Backend-Mechanismus ist einer von vielen, welche Ruby on Rails generiert, um den Arbeitsprozess zu beschleunigen.

Der PDF-Export

Ein wichtiges Tool für unsere Website war außerdem der Export von Schüler- und Bücherlisten als Druckfertige PDF-Dateien. Hierbei zeigte Ruby on Rails sich ein weiteres mal als Helfer, denn praktischerweise gibt es das RubyGem „Prawn“.

```
82  gem 'prawn', '~> 2.0', '>= 2.0.2'
83  gem "prawnto"
84  gem 'prawn-table', '~> 0.2.2'
```

Illustration 15: "Prawn" und "Prawn-Table" im Gemfile

„Prawn“ und „Prawn-table“ (https://github.com/prawnpdf/prawn) dienen dazu, im Controller daten aus der Datenbank auf eine PDF-Datei in einem neuen Tab zu schreiben. Moderne Browser können zum Glück die PDFs anzeigen und daher war diese Aufgabe mit Leichtigkeit zu bewältigen.

```
100                     pdf = Prawn::Document.new
101                     pdf.text "#{t.strftime("%Y-%m-%d")}"
102                     pdf.text "#{@student.full_name}"
103                     pdf.text "#{@student.classname.full_name}", size: 12
104                     pdf.text "#{@student.birth}", size: 10
105                     pdf.text "\n"
```

Illustration 16: Ruby-Prawn Code für unsere PDFs

Der gezeigte Codeausschnitt ist ein Teil der Funktion, die Listen von Schülern erstellt. Die gut leserliche Prawn-Syntax ist hierbei hilfreich.

In der Zeile 102 beispielsweise wird mit `pdf.text` angegeben, dass das folgende in die PDF geschrieben werden soll. Das darauf folgende `@student` gibt die Tabelle an, aus der der nächste Datensatz genommen weden soll und `full_name` gibt die Spalten mit dem Namen der Schüler weiter.

Das Resultat sieht dann wie folgt aus :

2017-07-02
TGJ1/2-2016

Schüler	Geburtsdatum	Preis
Fabius Quintus Balogh	1998-09-18	0.00€
Alexej Bredhauer	1998-05-09	0.00€
Kay David Stipcevic	1998-01-14	0.00€
Elias David König	2000-05-11	0.00€
Simon Kunz	1999-04-15	0.00€

Illustration 17: Ausschnitt aus der Preisliste der momentanen TGJ1/2

Das Generieren von Barcodes

Ähnlich wie beim Erstellen der Schülerlisten, mussten wir auch Barcodes generieren, welche als PDF exportiert werden. Diese werden dann auf den jeweiligen Büchern angebracht, um sie mit einem Barcodescanner auf der Oberfläche anzeigen lassen zu können. Zweck ist hierbei das schnelle Einsammeln und Aushändigen von Schulbüchern, sowie das Wiedererkennen von Buchkopien um Täuschungsversuchen zu entgehen.

Der Barcode repräsentiert lediglich die einzigartige ID einer jeden Kopie aus der „Copies“ oder Kopien Tabelle. Neben dem bereits erklärten Prawn-Gem zum Erstellen von PDF-Dateien, verwendeten wir hierbei noch das RubyGem „Barby“ (https://github.com/toretore/barby), das Barcodes zu eingegebenen Zahlenfolgen generiert.

```
94          barcode = Barby::Code128B.new(@copy.code)
95
96          require 'barby/outputter/prawn_outputter'
97          outputter = Barby::PrawnOutputter.new(barcode);
98
```

Illustration 18: Code zur Generierung der Barcodes in der PDF-Datei

Zunächst wird in Zeile 94 der Barcode deklariert, welcher wiedergegeben werden soll. Der Parameter `@copy.code` referenziert den Code, also die ID der jeweiligen Kopie in der Datenbank. Mit `Barby::Code1288.new` wird der Code im richtigen Format erzeugt.

Anschließend wird ein „Outputter“ gewählt. Der „Outputter“ bestimmt das Medium, auf welchem der Barcode wiedergegeben wird. Da wir es auf einer Prawn-PDF anzeigen wollen, wählten wir natürlich den „Prawn_Outputter“. Dieser hat nun nurnoch den bereits deklarierten barcode als Parameter und kann den Barcode auf der PDF anzeigen.

Auch hier waren die Gems von Ruby on Rails mehr als nur hilfreich. Vor allem das Korrelieren von „Barby“ und „Prawn“ erparte und einiges an Arbeit, da wir bereits vorher „Prawn“ für die Schülerlisten benötigten. Wir konnten also unser bereits Erlerntes direkt mit „Barby“ verknüpfen.

Illustration 19: "Hello World" als Barcode im Code128 Standard

Bedienungsanleitung:

Anlegen neuer Klassen:

Klassen → Neue Klasse → Daten eintragen → **edit class**

Anlegen neuer Schüler:

Schüler → Neuer Schüler → Daten eintragen → **create Student**

Anlegen neuer Bücher:

Bücher → Neues Buch → Daten eintragen → **create Book**

Bücher Schülern zuweisen / von Schülern einsammeln

Klassen → gewünschte Klasse **Anzeigen** → gewünschte Schüler **Anzeigen** → Stiftsymbol

Bücherlisten anzeigen:

Klassen → gewünschte Klasse **Anzeigen** → Bücherliste

Preislisten anzeigen:

Klassen → gewünschte Klasse **Anzeigen** → Preisliste

Fazit

Nach einem ganzen Schuljahr Arbeit als Gruppe am Projekt gibt es einige Schlüsse zu ziehen.

Die Auswahl unserer Technologie, sowie unsere ausgiebige Planungsphase waren das, was den Erfolg unseres Projektes ausmachten. Hätten wir das Java-Programm des vorherigen Jahres übernommen, wären wir wahrscheinlich immer noch nicht in der Lage, die Performanz bessern zu können. Nur mithilfe eines komplett neuen Projektes war dies möglich, denn wie bereits erwähnt haben wir mit Ruby on Rails und etwas Datenbankplanung die bisherigen Probleme komplett negieren können.

„Rails“ ermöglichte es uns außerdem durch die vielen Generatoren, einen relativ unaufwändigen Seminarkurs zu haben, allerdings war ein großer Teil unserer Arbeit das Erlernen von eben diesen Möglichkeiten.

Ein Problem während des gesamten Jahres war die Arbeitsaufteilung. Oftmals musste die gesamte Arbeit der Woche von ein bis zwei Mitgliedern der Gruppe gestämmt werden, da den restlichen das Know-How fehlte. Dies ließ einige im Schatten stehen, während andere, die sich intensiver mit der Materie auseinandersetzten, die Hauptarbeit erledigten.

Zum Abschluss bleibt nur noch zu sagen, dass die serverseitige Anwendung definitiv mehr Vorteile mit sich bringt als Nachteile und dass dies eine überlegene Lösung, im Vergleich zu einem nativen Programm ist. Sogar der Aufwand zur Programmierung wäre geringer gewesen, allerdings gestalteten einige gruppeninterne Probleme den Seminarkurs manchmal als äußerst schwierig. Seien es persönliche Diskrepanzen oder Demotivation, frequent wurde der „Work-Flow“ unterbrochen.

In Retrospektive war der Ursprung dieser Probleme wohl, dass wir uns untereinander relativ unbekannt waren, daher die anderen Gruppenmitglieder nicht vollkommen einschätzen konnten und jeder einzelne mehr oder weniger für sich arbeitete.

Quellen

https://www.couven-gymnasium.de/2014/09/bucherstapel-drohen-zu-verwaisen/

https://www.carolinabarcode.com/corded-barcode-scanner-types-a-68.html

https://www.c9.io

https://github.com/Hummeltron/SBV-AES

https://ide.c9.io/lord_gurke/kopieneinscannenundso

https://getbootstrap.com/components/#navbar