



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

PRAKTIKUMSBERICHT

Datenbanken und verteilte Systeme

Vorgelegt von:

Lennart Feldtmann 6449747

Tim Kilian 6824270

Sean-Patrick Malfa 6810393

Finn Rietz 6799896

Inhaltsverzeichnis

Abbildungsverzeichnis	i
1 Einleitung	1
2 Chat	4
2.1 Funktionalität	4
2.2 Probleme	5
3 Profil	7
3.1 Funktionalität	7
3.2 Probleme	8
4 Ranking	9
4.1 Funktionalität	9
4.2 Probleme	9
5 Account	11
5.1 Funktionalität	11
5.2 Probleme	12
6 Spiel	14
6.1 Funktionalität	14
6.2 Probleme	15
6.2.1 Baqend Real-Time vs. Socket.io	16
7 Technologien und Fazit	18

Abbildungsverzeichnis

Abb. 1.1	Vereinfachte Struktur des DB Entwurfs	3
Abb. 2.1	Spiel und Chat	4
Abb. 3.1	Profil	7
Abb. 4.1	Ranking Page	9
Abb. 5.1	Login Page vor dem Login	11
Abb. 5.2	Login Page nach dem Login	11
Abb. 6.1	Startgebiet des Spiels	14

1 Einleitung

In diesem Bericht gehen wir Schritt für Schritt darauf ein, wie unsere Anwendung funktioniert, welche Probleme sich uns in den Weg gestellt haben und wie wir diese dann beseitigt haben. Dabei unterteilen wir den Bericht in die einzelnen Komponenten “Game”, welche aus dem Spiel und dem Chat besteht, “Profil”, “Ranking” und “Account”.

Der zeitliche Ablauf (abgesehen vom Spiel, welches durchgängig über die 3 Wochen entwickelt wurde) ist erkennbar aus der Sortierung der Kapitel. Zuerst wollen wir aber auf die Ideenentwicklung der frühen Phase des Projekts und den daraus resultierenden Entscheidungen zu sprechen kommen.

Wir konnten uns schnell als Gruppe zusammenfinden, da wir zwei einzelne Zweiertteams waren und alle anderen bereits in ihren Gruppen aufgeteilt waren. Nachdem wir die Baqend Einführung bearbeitet haben, haben wir uns bereits am Dienstag Gedanken über unser Projekt gemacht. Wir waren uns schnell einig, dass wir ein Multiplayerspiel mit React implementieren möchten. Die Idee des Spiels kam aus unserem gemeinsamen Interesse an Spielen auch aus der Lust, einmal ein Spiel zu entwickeln. Für React hatten wir uns entschieden, da 3 von 4 Mitgliedern noch nie mit React gearbeitet haben und bei allen das Interesse bestand es zu lernen oder zu nutzen. Zudem bietet sich die Nutzung von React durch die Existenz des React-redux-starter-Projekts von Baqend an.

Wir haben schnell verworfen, das Spiel selbst mit React zu implementieren, da wir unser Spiel möglichst in Realtime laufen lassen wollten, haben React aber weiterhin für alles andere verwendet.

Als wir uns weitere Ideen für das Spiel überlegt hatten, haben sich die Interessen leicht getrennt. Ein paar von uns hatten die Idee geäußert ein “Open World” Spiel zu bauen, während die anderen der Meinung waren, dass wir mit einem beschränkten Canvas schon genug Aufgaben haben. Da wir agil vorgegangen sind, haben wir uns darauf geeinigt, es abhängig von der Zeit zu machen, die uns über bleibt, sobald die Grundfunktionalität des Spiels steht. Somit haben wir mit einem beschränkten Canvas gestartet.

Zunächst haben wir uns eine grobe Designskizze der Website erstellt. Neben dem Spiel wollten wir auch für jeden Spieler sein eigenes Profil haben und ein Ranking, indem sich die Spieler gegenseitig vergleichen können. Zudem waren wir uns einig, dass zu einem MMO ein Chat gehört, indem sich die Spieler gegenseitig Nachrichten zustellen können.

Für den Anfang haben wir nicht mehr geplant und haben angefangen, uns in die jeweilige Materie einzuarbeiten. Dabei haben wir uns die Aufgaben unterteilt und uns in den zwei Zweiergruppen vom Beginn des Praktikums wieder eingefunden. Finn und Sean-

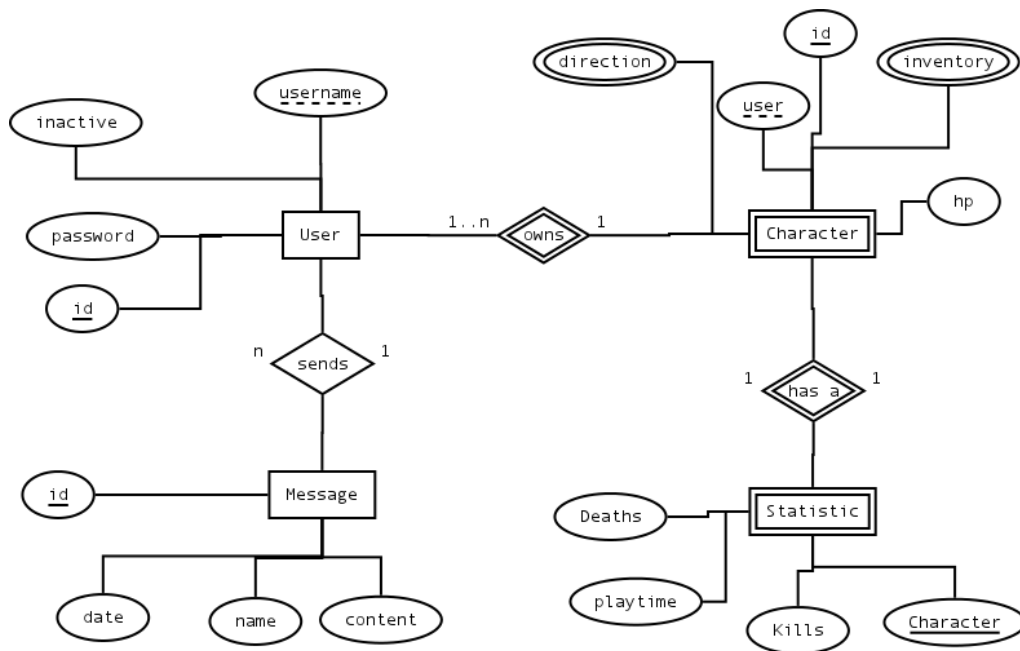
Patrick haben dabei die Aufgaben für den Chat, das Profil und das Ranking übernommen, während Tim und Lennart sich dem Spiel gewidmet haben.

Uns war es wichtig, die Grundfunktionalität der einzelnen Komponenten zu erreichen, weshalb wir versucht haben, uns so wenig Aufgaben wie möglich zur gleichen Zeit aufzugeben. Zudem haben wir Zusätze, die wir als nicht essentiell betrachtet haben, solange aufgeschoben, bis wir der Meinung waren, dass wir die wichtigste Funktionalität implementiert haben. Wir hatten als Gruppe zudem ab der zweiten Woche immer wieder die Diskussion, ob wir ein CSS-Framework benötigen oder ob wir alles per Hand machen können. Letztendlich haben wir ein CSS-Framework eingesetzt, welches “semantic-ui” heißt. Mit dem Einpflegen CSS-Framework sind nicht nur gute Effekte aufgetreten, da wir es erst relativ spät eingefügt haben und bereits viel für das Frontend gemacht haben und dies nun überarbeitet werden musste.

Als Backend haben wir uns für Baqend und Socket.io entschieden. Baqend übernimmt dabei die Verwaltung und Speicherung der für uns relevanten Daten. Dazu gehören zum Beispiel: User mit Name und Passwort, zugehörige Charaktere mit Eigenschaften wie Koordinaten auf der Karte, seinen Spielwerten (Vitality, Strength, Dexterity und Intelligence), aber auch spielexterne Informationen wie die geschriebenen Chatnachrichten, Statistiken.

Desweiteren gibt es mehrere Tabellen für Items: ItemMask, welches als Template für die Itemerzeugung dient. Item, welches erzeugte Gegenstände speichert. Equipment, welches die ausgerüsteten Gegenstände von Charakteren darstellt. Schließlich noch InventoryItem, welches die Gegenstände im Inventar des Nutzers hält, die er nicht direkt am Körper trägt. Zum besseren Verstehen haben wir ein ER-Diagramm erstellt, welches die Struktur in Baqend widerspiegelt. (Abbildung 1.1)

Im Folgenden betrachten wir die Komponenten im Einzelnen: Welche Funktionalität hatten wir uns gewünscht, welche wurde erreicht und welche Probleme sind dabei aufgetreten.

**Abbildung 1.1:** Vereinfachte Struktur des DB Entwurfs

2 Chat

2.1 Funktionalität

Der Chat liefert eine Echtzeitanbindung aller Spieler, die gerade aktiv in der Komponente “Game” sind. Er ist nicht für Spieler zugänglich, die sich in anderen Komponenten aufhalten (Account, Profil, Ranking). Der Chat updatet, sobald eine Nachricht geschrieben wurde und jeder Mitspieler kann diese sehen. Jeder Mitspieler hat natürlich die Möglichkeit, eine Nachricht zu schreiben. Bei jeder geschriebenen Nachricht wird zudem angezeigt, von wem sie verfasst worden ist. Jedesmal, wenn man die Komponente “Game” neu betritt, wird der Chat zurückgesetzt und man sieht keine Nachrichten, die geschrieben worden waren, bevor man in der Komponente “Game” aktiv war. Die letzte Funktion, die wir für den Chat implementiert haben, ist die Join-Info Nachricht. Diese war nicht so einfach, wie wir zuerst erwartet hatten. Auf den Vorgang der Funktion gehen wir näher bei den Problemen ein.



Abbildung 2.1: Spiel und Chat
Spiel und Chat sind vom Höhenverhältnis angepasst

Der Chat befindet sich rechts vom Spielfeld und ist mit einer Overflow-Scrollbar ausgestattet. Sollte es zu einem Overflow an Nachrichten kommen, so wird der Fokus immer auf die neusten Nachricht gelegt, während die alten Nachrichten unter der Scrollbar verschwinden.

Wir haben den Chat vom Spielfeld getrennt, damit die Steuerung nicht mit den Chatnachrichten in Konflikt treten. Somit kann man entweder im Chat schreiben oder Spielaktionen. Dies wird über den Fokus im Chatinput gesteuert.

Der Chat wird mit der Real-Time API von Baqend realisiert. Wir stellen eine Anfrage

an unsere Datenbank, welche uns alle Chatnachrichten gibt die verfasst wurden, seitdem wir in der Komponente Game sind. Hierbei handelt es sich um einen self-maintaining Query. Wir bekommen also jedesmal, wenn sich die Response auf unseren Query ändert, durch den verwendeten resultStream das neue Ergebnis übergeben. Konkret heißt das, dass jederzeit, wenn eine Nachricht von einer Person neu abgesendet wird, diese bei allen Usern als letzte Nachricht angezeigt wird.

2.2 Probleme

Beim Chat hatten wir ziemlich viele Probleme, da es unsere erste eigene Implementierung mit Baqend und React war. Um damit zurecht zu kommen, haben wir den Quickstarter von Baqend eingehend studiert, um zu verstehen, wie dieser Chat funktioniert. Währenddessen haben wir auch an dem Layout für den Chat gearbeitet. Unser Ziel war es, den Code aus dem Quickstarter Stück für Stück in unser React-Layout einzufügen. Da der Quickstarter nicht mit React realisiert wurde, war die Semantik anders und nicht für unser Projekt geeignet.

Wir hatten es jedoch geschafft, die sendMessage Funktion zu unseren Gunste umzuschreiben. Zudem haben wir es recht schnell geschafft, einen Datenbankzugriff aufzubauen. Als wir nun also unsere Nachrichten in der Datenbank hatten, war unser nächstes Problem, wie wir sie abrufen können. Wir haben uns dabei auch am Quickstarter orientiert, mussten es aber verwerfen, als wir gesehen haben, dass unsere Nachrichtenanzeige sich nur dann aktualisiert, wenn wir selbst eine Nachricht schreiben und versenden und nicht dann, wenn auch jemand anderes eine Nachricht schreibt. Deshalb mussten wir auf die Baqend Real-Time API umsteigen. Wir kannten uns noch nicht mit den Real-Time Queries aus, weshalb wir hier viel Hilfe in Anspruch nehmen mussten. Die kam meistens von Julian, da einige Inhalte für Baqend/React noch in Arbeit waren, als wir mit diesen arbeiten wollten. Als der Chat so lief, wie wir uns diesen vorgestellt haben, mussten wir die Optionen ausschließen, mit denen Fehler erzeugt werden können oder unlogische Handlungen vollzogen werden.

Dazu haben wir Baqend-Handler eingefügt. Dies hat nicht ganz so funktioniert, wie wir uns das vorgestellt haben. Der Handler, welcher für leere Nachrichten verantwortlich ist, funktionierte einwandfrei, während der Handler für den nicht eingeloggten User nicht funktionierte. Wir schließen zwar durch unseren Code aus, dass jemand, der nicht eingeloggt ist eine Nachricht schreiben kann, jedoch wird keine Fehlermeldung dadurch geworfen. Wir haben uns letztendlich dazu entschieden, den Chat nicht zu rendern, wenn der User nicht eingeloggt ist.

Ein weiteres Problem war unsere gewünschte Funktion der join-Nachricht. Wir hatten

bereits eine solche Nachricht implementiert, jedoch hat uns diese nicht gefallen, da sie bei jedem betreten der “Game” Komponente diese Nachricht geschrieben hat. Wenn jemand oft hin und her gesprungen ist, so wurde die Datenbank mit dieser Nachricht überflutet und dies war nicht unsere Absicht mit der Nachricht. Aus diesem Grund haben wir diese Nachricht vorerst herausgenommen. Unser Plan war es dann, diese Nachricht über die App.js geben zu lassen, da diese nur einmal aufgerufen wird, und zwar genau dann, wenn man das Spiel startet. Somit hätten wir dann die Möglichkeit gehabt, diese Nachricht einmal nach dem Einloggen zu geben und danach nicht immer wieder neu zu schreiben. Jedoch konnten wir dies nicht wie geplant durchführen, da wir dann mehrere exports gehabt hätten und dies zu einem Konflikt führte. Unser letzter Versuch war dann, diese Nachricht schreiben zu lassen, wenn sich die Person einloggt oder registriert, da dies ein einmaliges Event ist und jeder Spieler dies machen muss, um den Chat überhaupt nutzen zu können. Dies hat unser Problem gelöst.

Das letzte Problem, welches wir überwinden mussten (von welchem wir lange gar nicht mitbekommen hatten, dass es existierte), stellten die Subscriptions dar, welche mit der Real-Time API von Baqend zusammenhängen. Immer wenn wir einen Real-Time-Query erstellen, beginnt dieser zu beobachten, ob sich sein Ziel-Objekt ändert. Irgendwann fiel uns auf, dass eine Fehlermeldung geworfen wurde, wenn wir sehr häufig aus der Game Komponente heraus und wieder hinein wechselten, welche besagte, dass wir zu viele Subscriptions hätten. Dies lag daran, dass wir in `componentWillMount` jedes mal eine neue Subscription erzeugten, wenn wir die Komponente Game betraten, ohne diese jemals wieder zu entfernen. Dadurch entstanden mehrere, verschiedene Observer, was zu sehr verwirrendem Verhalten im Chat führte, da es völlig zufällig schien wann nun welche Nachrichten angezeigt werden. Dieses Problem konnten wir dann beheben indem wir in `componentWillUnmount` wieder unsubscribe.

3 Profil

3.1 Funktionalität

Das Profil bietet eine bildliche Darstellung des eigenen Charakters. Hier werden sowohl die Charaktereigenschaften und Statistiken angezeigt, als auch die Ausrüstung im Inventar. Das Profil updatet sich nur beim erneuten Laden, nicht in Real-Time. Die Statistiken beziehen sich nur auf den eingeloggten Spieler.

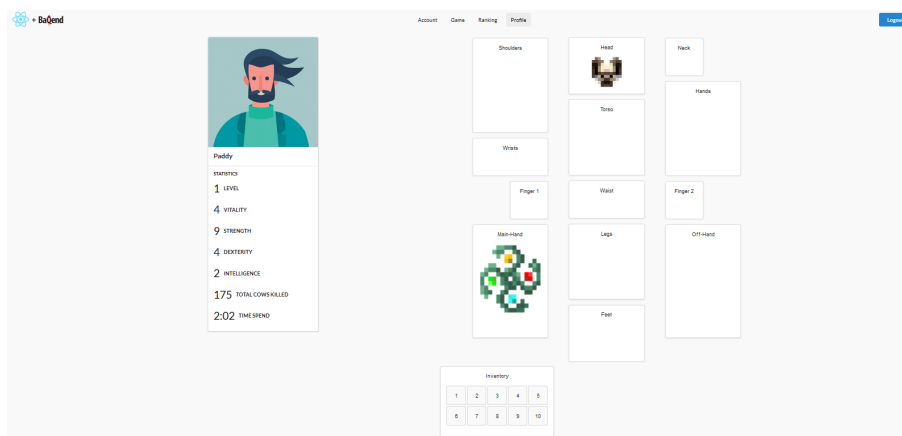


Abbildung 3.1: Profil

Nur von eingeloggten Nutzern sichtbar. Diese sehen auch nur ihr eigenes Profil in dieser Form

Es werden die Charaktereigenschaften Vitality, Strength, Dexterity, Intelligence und Level angezeigt, als auch die Statistiken Total Cows killed und die gespielte Zeit. Wir unterscheiden dabei zwischen Charaktereigenschaften und Statistiken.

Charaktereigenschaften sind all diese, welche für das Spiel relevant sind, da sie für Berechnungen notwendig sind oder sein werden. Statistiken ergeben sich aus den Ereignissen im Spiel, werden aber nicht weiter im Spiel verwendet. Ein Beispiel ist die Anzahl der erlegten Kühe.

Diese Werte werden unterhalb des Profilbildes angezeigt. Die Werte sind entweder unter der Statistik oder dem Charakter in der Datenbank gespeichert und werden mit simplen Querys aus jener abgefragt.

Der Hauptteil der Seite besteht aus der bildlichen Darstellung des Spielers. Hierbei haben einzelne Körperteile oder Regionen eine eigene Darstellung, welche mit gesammelten Items ausgestattet werden können. Auf dem Bild wurde für den Kopf ein Helm ausgestattet und für die Hand ein Shuriken. Wenn man mit der Maus über eines der Items fährt, so erfährt man die Werte, die dieses Item hat. Diese werden zu den Charaktereigenschaften

hinzugerechnet und nehmen Einfluss ins Spiel.

Gegenstände, die man zwar aufgesammelt, jedoch nicht ausgerüstet hat, sollen unterhalb der bildlichen Darstellung in der Tabelle “Items” angezeigt werden. Diese Funktionalität ist noch nicht gegeben, planen wir jedoch nach dem Praktikum einzubauen. Zudem sollen sie mithilfe eines Drag-and-Drop Systems ausgetauscht werden können.

3.2 Probleme

Das Profil haben wir zeitlich nach dem Chat implementiert.

Unser größtes Problem war die richtige Abfrage der Statistik Entitäten, da unsere Entität „Statistik“ als Schlüssel einen „Charakter“ hat, welcher wiederum als Schlüssel einen User hat. Also mussten wir zuerst die richtige User Entität bekommen um dann die richtige Character Entität zu haben.

Dieses Problem haben wir dadurch gelöst, indem wir die Anfrage immer leicht angepasst haben, bis wir das richtige Ergebnis erhalten haben. Schließlich war es eine `.then(..)` Abfrage, die uns gefehlt hat um ein Promise aufzulösen.

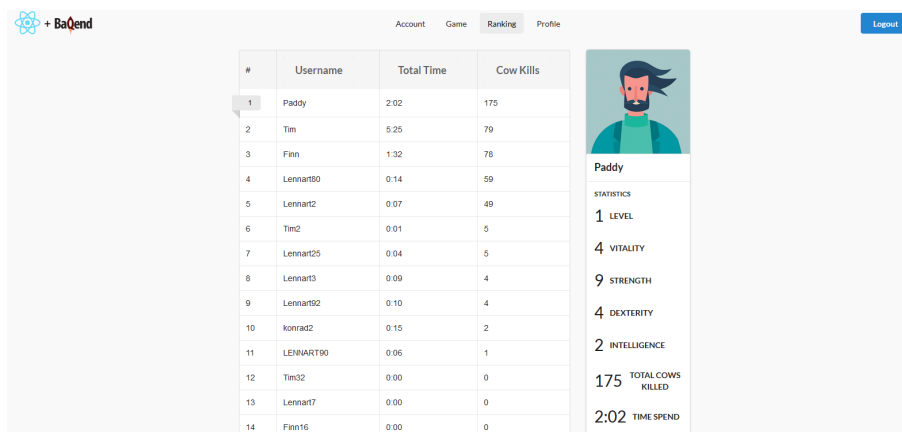
In der letzten Woche mussten wir, nach dem Einsatz von semantic UI, das Profil ganz neu gestalten, was uns einige Zeit gekostet hat. Jedoch sind dabei keine weiteren Probleme aufgetreten.

Im Verlauf des Projektes haben wir uns zudem entschieden, das Spiel primär als PvE (Player vs Enemy) laufen zu lassen, weshalb wir uns dann auch entschieden haben, andere Werte in der Statistik anzeigen zu lassen. Vor der Änderung hatten wir auch Werte gespeichert wie die Anzahl der Tode oder dem Kills/Tode-Verhältnis.


4 Ranking

4.1 Funktionalität

Das Ranking stellt die Möglichkeit dar, seine Statistiken mit anderen Spielern zu vergleichen. Zur Entität Statistik gehören jedoch lediglich die Attribute “cows killed” und “time spend”. Aus diesem Grund kann man in der Statistik auch nur nach den getöteten Kühen, der Zeit und dem Usernamen sortieren. Die einzelnen Spalten lassen sich mit jedem Klick auf- oder absteigend ordnen. Die Standardsortierung sind die Kills in absteigender Folge. Bei Bedarf kann man auf jede einzelne Zeile der Tabelle klicken und wird mit weiteren Charakterinformationen der anderen Spieler versorgt. Standardmäßig werden hier die eigenen Informationen angezeigt. Diese Informationen gleichen sich derer, die im Profil angezeigt werden.



#	Username	Total Time	Cow Kills
1	Paddy	2:02	175
2	Tim	5:25	79
3	Finn	1:32	78
4	Lennart90	0:14	59
5	Lennart2	0:07	49
6	Tim2	0:01	5
7	Lennart25	0:04	5
8	Lennart3	0:09	4
9	Lennart92	0:10	4
10	Konrad2	0:15	2
11	LENNART90	0:06	1
12	Tim32	0:00	0
13	Lennart7	0:00	0
14	Finn16	0:00	0



Paddy

STATISTICS

1 LEVEL

4 VITALITY

9 STRENGTH

4 DEXTERITY

2 INTELLIGENCE

175 TOTAL COWS KILLED

2:02 TIME SPEND

Abbildung 4.1: Ranking Page

Links die Tabelle mit den sortierten Statistiken, Rechts das ausgewählte Profil, in diesem Fall ist es das Profil des eingeloggtten Nutzers

4.2 Probleme

Das Ranking wurde zeitlich nach dem Chat und dem Profil implementiert. Es sind wenig Probleme aufgetreten. Da wir bereits die Profile implementiert hatten, war dieser Teil des Rankings schon fertig.

Die Abfrage auf die Datenbank hat für uns inzwischen auch kein Problem mehr dargestellt. Das einzige Problem bestand darin, dass wir beim Klicken auf ein Profil auch die richtigen Werte für das Profil anzeigen lassen. Als wir unser Ranking als “Table” realisiert hatten fehlte nur noch eine Funktion, mit der wir über alle Inhalte iterieren und überprüfen, auf welche Zeile geklickt worden war.

Als wir in der letzten Woche noch semantic UI als CSS-Framework hinzugefügt hatten, haben wir zunächst auch die Funktionalität des Rankings und der Profilanzeige zerschossen. Also mussten wir diese erst wieder herstellen, was uns mehr Schwierigkeiten bereitet hat als erwartet, da es sich mit einem Framework deutlich anders arbeiten lässt als ohne. Die Darstellung unterschied sich jeweils deutlich. Aus diesem Grund mussten wir uns erstmal verstehen, wie die Darstellung mit semantic UI funktioniert, um unseren bereits vorhandenen und funktionierenden Code so umzubauen, dass er auch mit dem Framework funktioniert.

5 Account

5.1 Funktionalität

Die Account Komponente liefert die Funktionen registrieren, einloggen und ausloggen.

Auf dem oberen Bild ist zu erkennen, dass eine Person, die nicht eingeloggt ist, nicht auf die Komponente Profil zugreifen kann. Da das Profil über den eingeloggten Nutzer generiert wird, haben wir uns dazu entschieden, das Profil nicht anzeigen lassen zu können, wenn man nicht eingeloggt ist. Das Profil wäre ansonsten bloß mit vielen “null” und “NaN” Werten gefüllt. Wie auf dem unteren Bild zu sehen ist, wurde nach dem Login die Komponente “Profile” zur Navigationsleiste hinzugefügt.

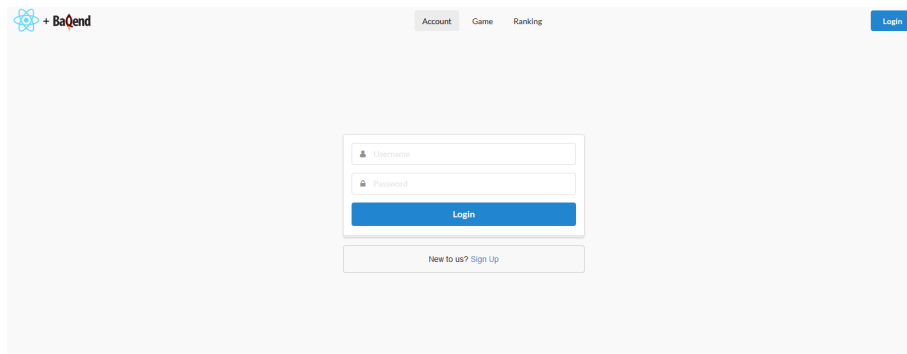


Abbildung 5.1: Login Page vor dem Login
Der Zugang zum Profil ist verwehrt

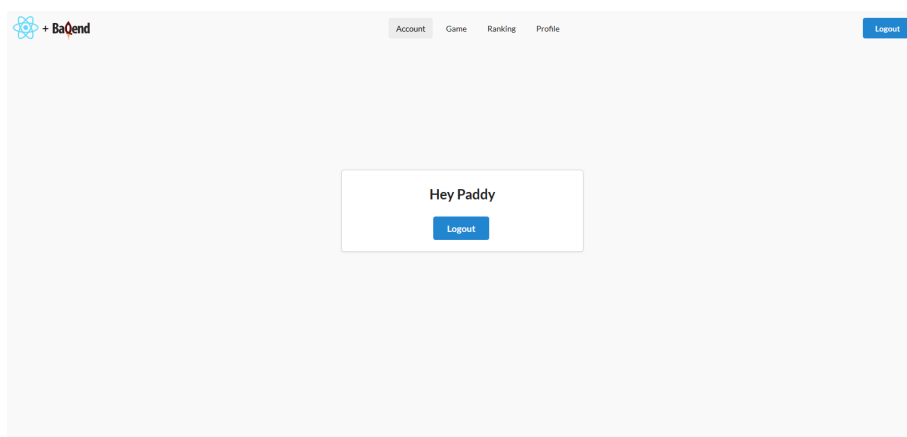


Abbildung 5.2: Login Page nach dem Login
Hier hat der eingeloggte Nutzer Zugang zu allen Komponenten des Spiels

Sollte eine Person zum ersten Mal die Seite besuchen und ist noch nicht im Besitz eines Accounts, so kann dieser seine Daten ganz normal in die Felder eingeben. Anstatt auf “Login” zu drücken, klickt man auf “Sign Up”.

Sollte der Username noch nicht vergeben worden sein und die Anzahl für den Username weniger als 10 Alphanumerische Zeichen betragen, so wird der Nutzer mit seinen Daten registriert und auf den Bildschirm des eingeloggten Nutzers geführt.

Sofern jedoch mindestens eine der beiden Bedingungen nicht erfüllt ist, so wird der Person eine Fehlermeldung ausgegeben, welche ihn darauf hinweist, welche Bedingung nicht erfüllt ist.

Sollte man bereits einen Account haben und versuchen sich einzuloggen, so werden die Bedingungen geprüft, ob der Username in der Datenbank existiert und ob das Passwort mit dem Username zusammengehört. Auch hier wird dem Nutzer im Fehlerfall entsprechende Meldung ausgegeben.

5.2 Probleme

Im Wesentlichen gelang es uns recht gut die grundlegende Funktionalität des Logins, Registrierens und Logouts zu implementieren. Diese wurden mit dem React/Redux Starter mitgeliefert.

Deutlich schwerer fiel es uns jedoch die Handler von Baqend richtig zu benutzen und Fehler, welche von Baqend geworfen werden, richtig abzufangen.

Wir waren zu Beginn des Praktikums noch überhaupt nicht damit vertraut, was eigentlich passiert, wenn man versucht sich mit einem neuen Nutzernamen zu registrieren, der schon in der Datenbank existiert. Aus diesem Unwissen heraus hatten wir ein neues Modul erstellt (welches man noch unter dem Baqend-Code-Reiter im Dashboard anschauen kann), welches überprüfen sollte, ob ein Benutzername bereits in der Datenbank existiert, oder nicht.

Abhängig von diesem Modul hätten wir dann die Registration abgelehnt oder zugelassen. Dies brachte weitere Probleme mit sich, weil jetzt Node Query-Rechte benötigte um überhaupt die Anfrage an die Datenbank stellen zu können.

Im finalen Stadium unserer App benutzen wir dieses Modul nicht mehr. Deswegen haben wir Node mittlerweile die Query-Rechte auch wieder entzogen. Wir zeigen stattdessen die Fehlermeldung an die wir von Baqend bekommen, sollte ein Benutzername bereits registriert sein oder falls das Passwort inkorrekt ist.

An einer Stelle benutzen wir jedoch noch eine eigene Fehlermeldung, beim Prüfen auf die Validität des User-Inputs. Sollte ein User nichts eingeben und auf Login klicken, so bekommen wir von Baqend die Nachricht “the body cant be processed”. Diese war uns nicht aussagekräftig genug, um sie dem Nutzer direkt anzuzeigen. Daher zeigen wir eine eigene Nachricht an.

Ein anderes Problem war, dass es zu Inkonsistenzen zwischen dem Frontend und Backend kommen konnte. Ursprünglich war der Plan, die Input-Fields zurückzusetzen, sobald ein es zu einer Fehlermeldung kommt. Im Backend war es nicht schwer umzusetzen, allerdings sind wir daran gescheitert, die Input-Fields im Frontend tatsächlich zurück zu setzen.

So konnte es dazu kommen, dass im Frontend der Nutzernamen noch angezeigt wurde, im Backend war das Feld welches den Nutzernamen hält jedoch schon auf einen leeren String zurückgesetzt. Da dies zu hoher Verwirrung beim Benutzer führen kann, haben wir uns für ein Workaround entschieden, indem wir die Input-Fields nur dann zurücksetzen wenn die Komponente gewechselt wird, da dies mit dem automatischen zurücksetzen der Input-Fields im Frontend einher geht und unserer Funktionalität nicht schadet.

6 Spiel

6.1 Funktionalität

Das Spiel ist ein MMO (Massiv Multiplayer Online) Game, welches wir trotz anfänglicher Skepsis gegenüber des Implementationsaufwands als “Open World” realisieren konnten.

Die Steuerung geht primär über die Pfeiltasten, welche zur Bewegung des Charakters dienen und der S-Taste, welche zum feuern der Faust benötigt wird. Andere Gegenstände werden über die Zahlentasten ausgelöst.

Jedem Nutzer, der sich registriert, wird ein Character und eine Statistik auf Baqend erstellt. Diese enthalten Spielinformationen wie Koordinaten auf dem Spielfeld, ob der Charakter sich gerade im Spiel befindet und viele weitere Attribute.



Abbildung 6.1: Startgebiet des Spiels

In der oberen Hälfte des Kartenausschnittes ist die Stadt zu erkennen, unterhalb ist die Kuhzone, in der die Kühe leben

Wenn der Charakter im Spiel ist, so landet er auf seiner Initialposition, welches im Stadt in der Kartenmitte ist. Der Nutzer sieht seinen Charakter immer im grünen Hemd, während alle anderen in einem roten Hemd angezeigt werden (siehe Abbildung).

Von hieraus kann er sich frei über das Spielfeld bewegen, wobei die Ansicht immer auf ihn fokussiert ist. Man hat nun die Möglichkeit, andere Spieler und Kühe anzugreifen und ihnen Leben abzuziehen. Wenn die Lebensanzeige auf 0 und weniger fällt, so wird das jeweilige Objekt bei keinem Client mehr angezeigt.

Die Lebensanzeige wird als roter Balken über dem Kopf angezeigt. Jedoch wird diese nur dann für alle Clients sichtbar, wenn die Lebensenergie nicht mehr bei 100 Prozent liegt. Sollte dieses Objekt eine Kuh gewesen sein, so lässt sie immer einen zufälligen Gegenstand fallen, welcher in das Inventar des Sammlers eingetragen wird.

Jedoch lassen sich Kühe nicht einfach erlegen, da sie mit einem Aggrosystem ausgestattet sind, sie kämpfen also gegen ihren Angreifer und können ihm auch Schaden zufügen. Momentan fügen sie ihrem Angreifer bei jeder Kollision einen Schadenspunkt zu.

Stirbt ein Charakter, wird dieser nach 5 Sekunden an seiner ursprünglichen spawn position wiederbelebt. In der Zwischenzeit folgt die Kamera einem anderen Spieler oder notfalls einer Kuh, wenn keiner vorhanden ist.

Das Spiel läuft in 40 FPS um für smooth scrolling zu sorgen.

Die Karte ist mit einer Collisionmap ausgestattet, mit der verhindert wird, dass ein Nutzer aus der Spielwelt herausgehen kann und es gibt weiteren Elemente in der Welt, wie zum Beispiel Häuser und Tore mit Kollision. Momentan ist es dem Spieler jedoch noch möglich, die Spielwelt nach rechts zu verlassen.

Wir pflegen den Gedanken, in Zukunft weitere Teile der Karte einzufügen, welche über eben diese Lücke am Kartenrand erreichbar sein werden sollen. Für die Zukunft nach dem Praktikum haben wir zudem noch vor, ein Quest-System einzubauen, um dem Erlegen der Kühe einen tieferen Sinn zu geben.

6.2 Probleme

Auch wenn wir in unserer momentanen Implementierung für möglichst viel Robustheit sorgen wollten, gibt es noch das ein oder andere ungelöste Problem.

Bekannte Bugs sind:

- Collision funktioniert nicht immer einwandfrei, die Fäuste treffen teilweise nicht ihre Ziele.
- Chrome wirft manchmal Fehler, die ebenfalls Collision bedingt sind, und das Spiel lahmlegen
- Firefox stürzt in seltenen Fällen ab.
- Beim wechseln von Komponenten wird manchmal der eigene Charakter nicht korrekt ausgewählt.

Diese Bugs sind leider schwer zu debuggen, und sind teilweise abhängig vom Browser und Fehlern im verwendeten Collision-Framework. Da dieses Framework dennoch die beste Lösung für create.js liefert, haben wir an diesem festgehalten.

6.2.1 Baqend Real-Time vs. Socket.io

Lange Zeit stand nicht fest, wie wir die Daten innerhalb des Spiels austauschen wollen. Für die persistente Speicherung von Items, Statistiken etc. war von Beginn an klar, dass wir Baqend nutzen wollen. Sowohl die Leistungen der Technologie als auch die Möglichkeit, direkt mit dem Entwicklerteam zu kommunizieren haben uns sehr zugesagt. Da Baqend eine sehr niedrige Latenz liefert, wurde unsere Idee ein Spiel zu bauen, noch mehr bestätigt.

Zunächst hatten wir einen Server aufgesetzt, auf dem Socket.io lief. Da dieser Vorgang gut dokumentiert war und wir den Server zudem kostenlos zur Verfügung hatten war dies relativ einfach und schnell zu realisieren.

Nach der ersten Präsentation wurde uns die Real-Time API vom Baqend-Team nahegelegt, über die wir das Spiel laufen lassen könnten. In diese haben wir uns folglich eingelesen und eine Verbindung zwischen Spielern über subscription auf eventStreams hergestellt. Diese Verbindung besteht bis heute.

Immer wenn ein Spieler aufhört, alle Tasten zu drücken, wird ein Update mit allen wichtigen Informationen über seine Spielfigur an Baqend geschickt.

Für ein gutes Multiplayererlebnis mussten wir mindestens ein Update bei jedem Tastendruck senden, da sonst wichtige Änderungen an der Spielfigur, wie beispielsweise die Laufrichtung, nicht an alle Mitspieler kommuniziert werden. Außerdem wollten wir ein Echtzeitspiel erstellen, bei dem rapide Änderungen von Tastendrücken nicht verhindert oder abgefangen werden.

Diese Anforderungen führte bei der Nutzung von Baqend Real-Time zu Problemen. Wird ein Objekt gerade beschrieben, setzt Baqend clientseitig ein Lock auf dieses. Wird nun in sehr kurzem Abstand mehr als eine Änderung an einem Objekt getätigt, können nicht alle diese Änderungen getätigt werden, wodurch Informationen verloren gehen. Dieses Problem konnten wir leider auch nicht mit Hilfe des Baqend-Teams lösen, sodass wir uns schließlich für einen Gameserver mit Socket.io entschieden haben.

Socket io basiert auf serverseitigen und clientseitigen Quellcode, weshalb wir unseren Gameserver erstmal irgendwo bereitstellen mussten. Ein uns bereits bekannter Anbieter, DigitalOcean, bot uns hierfür eine einfache Lösung. In sehr kurzer Zeit hatten wir einen lauffähigen Linux-Server, den wir daraufhin mit Nginx konfiguriert haben um einen Node zu starten auf dem Socket io lief.

Socket besteht im Grunde nur aus zwei wichtigen Funktionen, die on und die emit Funktion. Emit kann ein Javascript-Object von Client zu Server und Server zu Client senden. On ist der passende Listener dazu, der auf die gesendete Aktion reagiert. Mit Socket war

es uns theoretisch sogar möglich, auf jedem Tick des Spiels ein Objekt zu senden, also 30 mal die Sekunde. Verluste der Datenpakete haben wir dabei keine bemerkt.

Charakterupdates werden also an den Server gesendet. Dieser reagiert darauf mit einem Broadcast an alle anderen Spieler mit dem erhaltenen Objekt. Zudem sorgt der Server für die Erzeugung und AI der Tiere auf der Karte, damit diese für alle Spieler identisch sind.

Mit der Performanz unseres Servers sind wir zu 95% zufrieden. Eine noch bessere Anbindung wäre vielleicht mittels WebRTC möglich gewesen, dafür hätten wir allerdings Zeit aufgeben müssen, die wir in unser Spiel investieren mussten. In der Zukunft schauen wir uns diese Möglichkeit wohl nochmal genauer an.

7 Technologien und Fazit

Folgende Technologien haben wir bei der Entwicklung unserer App verwendet:

- Baqend: Real-Time API für den Chat und das Spiel, normale Queries für Profil und Statistik
- React: Baqend/react-redux-starter für die Navigation und Funktionalität der Website
- Semantic UI: Für das Stilisieren der Webpage
- Socket.io: Für die Echtzeit-Verbindung zwischen Spielern
- CreateJS: Easel.js zum bearbeiten von Sprites auf dem HTML5-Canvas und Tween.js für gleitendere Bewegung der Kamera
- Collision-Detection: Eine Erweiterung zu Create.js zum Feststellen von Kollisionen auf dem Canvas

Insgesamt sind wir mit unserem Erreichten sehr zufrieden. Obwohl unsere Ansprüche mit der Zeit immer weiter gewachsen sind, da wir ständig neue Ideen hatten, haben wir unsere anfangs gesetzten Ziele alle erreicht. Unser aktuelle App ist größtenteils robust, unser Code hat eine unserer Meinung nach anständige, wenn auch nicht immer perfekte Struktur und wir haben auf sinnvolle Weise Technologien bzw. Frameworks in unser Projekt integriert.

Natürlich haben wir trotz etwas Vorerfahrung viel über die eingesetzten Frameworks, Javascript und Datenbanken/Server gelernt, doch im Vorrang nehmen wir für uns persönlich die Erfahrungen aus der Arbeit im Team an einem gemeinsamen Projekt mit. Dazu gehören die faire Abstimmung von Entscheidungen, die gegenseitige Hilfe, regelmäßiges Pair-Programming, Planen und Entwerfen eines Projekts, Zeitmanagement und vieles mehr. Den Nutzen all dieser Dinge bekommt man zwar im Studium vermittelt, richtig lernen und erfahren tut man dies erst bei der Arbeit an einem handfesten Produkt wie im DuVS-Praktikum.