Autoren: Danny, Mike, Julian

Beschreibung des Projekts

Entwickelt werden soll ein Idle-Clicker-Spiel, welches hohe Benutzerfreundlichkeit, hohe Content-Flexibilität und ebenfalls eine hohe Spielbarkeit mit viel Unterhaltsamkeit bietet.

Den Kern eines Clicker-Spiels bildet ein klickbares Objekt, in diesem Fall ein Computer, welcher angeklickt werden muss, um eine Währung **„BitCoins“** zu generieren. Mittels dieser Währung können dann Gebäude erworben werden, welche sowohl das Passiveinkommen der Währung pro Sekunde erhöhen als auch eine Erhöhung des Ertrages durch das Klicken des klickbaren Objekts.

Neben den Gebäuden gibt es außerdem Upgrades, welche entweder die Produktion eines Gebäudes oder die Gesamtproduktion erhöhen können.

Ist der Spieler offline, so wird weiterhin Einkommen generiert, u. a. um einen höheren Anreiz zum Wiederöffnen des Spiels zu bieten.

Ziel des Spiels ist es, durch den Kauf möglichst vieler Gebäude und Upgrades möglichst hohe Summen an Einkommen und Währung zu generieren; diesen Zahlen ist keine realistische Grenze gesetzt, sodass das Spiel als endlos zu betrachten ist.

Spielbar ist das Spiel im Browser.

Nutzerhandbuch

Klickbares Objekt

Unter dem „klickbaren Objekt“ wird hier der Computer verstanden, der sich mittig im Clicker-Frontend befindet. Dieser dient dazu, beim Klick als Aktiveinkommen die Währung BitCoins zu generieren.

Währung

Die Währung, gemessen in BitCoins, dient dazu, Gebäude und Upgrades zu kaufen, erworben wird sie durch Klicks auf das klickbare Objekt sowie durch Passiveinkommen.

Aktiveinkommen

Aktiveinkommen ist das Einkommen, das durch direkte, vom Benutzer durchgeführte Klicks auf das klickbare Objekt generiert wird. Es kann durch Gebäude und Upgrades erhöht werden.

Passiveinkommen

Passiveinkommen wird in „**BitCoins pro Sekunde**“ gemessen. Folglich wird die Währung jede Sekunde um diese Menge an BitCoins erhöht. Das Passiveinkommen bestimmt ebenfalls die Menge an Einkommen, welche Offline generiert werden kann.

Gebäude

Die Liste der kaufbaren Gebäude befindet sich links im Clicker-Frontend. Gebäude dienen dazu, die Produktion von Währung sowohl beim Passiveinkommen als auch Aktiveinkommen zu erhöhen.

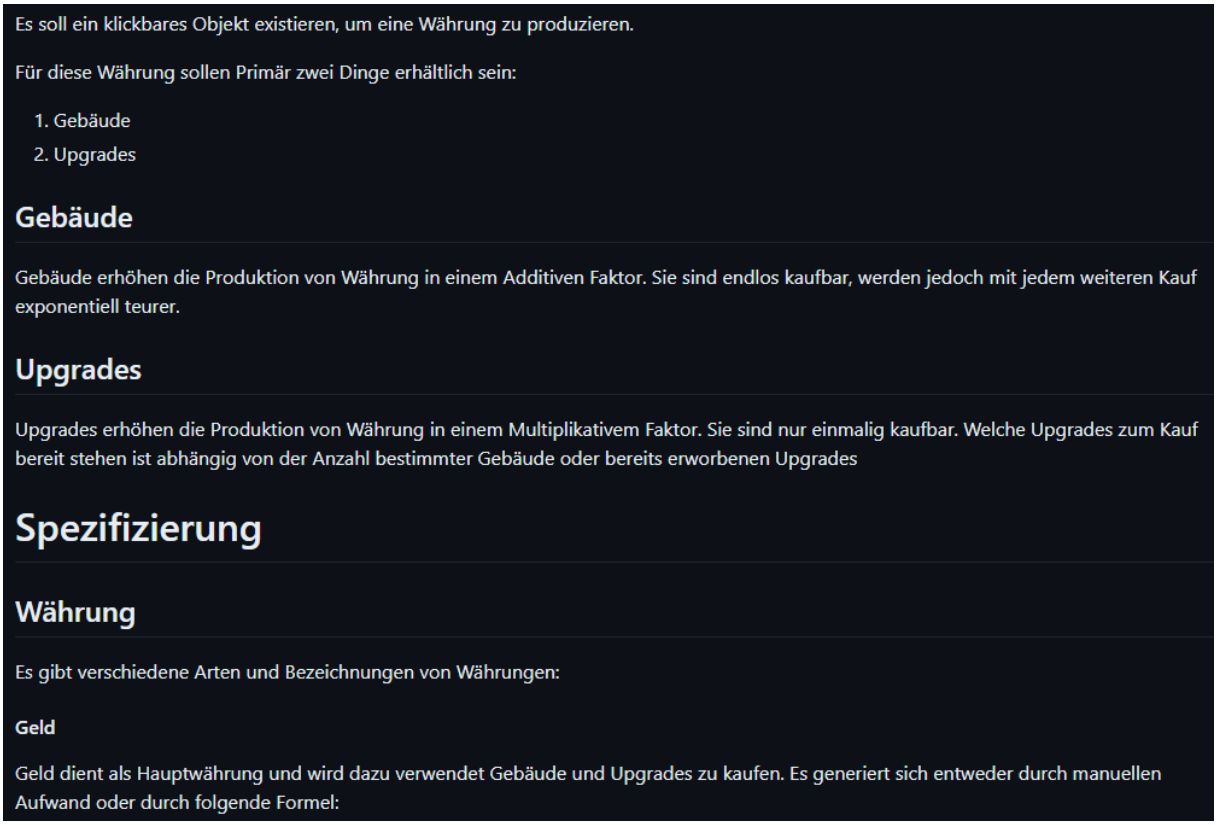
Upgrades

Die Liste der kaufbaren Upgrades befindet sich rechts im Clicker-Frontend. Upgrades dienen dazu, die Produktion von Gebäuden zu erhöhen, oder die Gesamtproduktion (sowohl pro Click wie auch pro Sekunde) zu erhöhen.

Projektablauf

Erste Planung

In der Planungsphase wurden zunächst die Fragen der Arbeitsmethodik geklärt. So haben wir uns darauf geeinigt, Anforderungen im Behavioral Specifications Format zu schreiben. Außerdem haben wir bei der Planung das Kanban-Board über GitHub Projects eingerichtet. Daraufhin haben wir mit der Anforderungsanalyse im Markdown-Notizen-Tool Obsidian begonnen:


Es soll ein klickbares Objekt existieren, um eine Währung zu produzieren.

Für diese Währung sollen Primär zwei Dinge erhältlich sein:

1. Gebäude
2. Upgrades

Gebäude

Gebäude erhöhen die Produktion von Währung in einem Additiven Faktor. Sie sind endlos kaufbar, werden jedoch mit jedem weiteren Kauf exponentiell teurer.

Upgrades

Upgrades erhöhen die Produktion von Währung in einem Multiplikativem Faktor. Sie sind nur einmalig kaufbar. Welche Upgrades zum Kauf bereit stehen ist abhängig von der Anzahl bestimmter Gebäude oder bereits erworbenen Upgrades

Spezifizierung

Währung

Es gibt verschiedene Arten und Bezeichnungen von Währungen:

Geld

Geld dient als Hauptwährung und wird dazu verwendet Gebäude und Upgrades zu kaufen. Es generiert sich entweder durch manuellen Aufwand oder durch folgende Formel:

Nach der ersten Planung wurden dann die Anforderungen geschrieben und die Umsetzung begonnen.

Umsetzung des Backends

Das Backend basiert auf einer REST-API und einer PostgreSQL-Datenbank. Die Rest-API wurde mithilfe von Rust entwickelt und beide sind, wie das Frontend auch, containerisiert. Dazu haben wir uns entschieden, die Applikation plattformagnostisch und möglichst portabel zu gestalten. So muss somit nur Docker installiert werden, um die gesamte Applikation auf einem Gerät zu starten.

Des Weiteren ist ebenso für die Entwicklung am Backend ein Dev-Container aufgesetzt worden, mit dem auch andere Entwickler problemlos am Projekt mitwirken können, was eine Bearbeitung dererseits vereinfacht. Mithilfe eines Dev-Cotainers lässt sich ebenso die API

möglichst produktionsnah testen und bearbeiten, da lediglich umgebungsspezifische Konfigurationen wie Umgebungsvariablen unterschiedlich sind.

Die REST-API wurde mithilfe von Actix aufgesetzt. Bei Actix handelt es sich um ein Rust-Crate (Package), welches die Entwicklung von schnellen und skalierbaren Webanwendungen durch asynchrone Verarbeitung, eingebauten HTTP/2-Support ermöglicht und eine strukturierte, ergonomische API beinhaltet, die das Erstellen von RESTful APIs erleichtert. Actix bietet zudem umfangreiche Middleware-Unterstützung und eine starke Typisierung für eine robuste und sichere Codebasis.

Für die Anbindung an die Datenbank hat man sich für Diesel entschieden. Dies ist ein leistungsstarkes ORM (Object-Relational Mapping) für die Rust-Programmiersprache, das die Interaktion mit relationalen Datenbanken vereinfacht. Es ermöglicht das Definieren von Datenbankmodellen durch Rust-Strukturen und bietet eine sichere und komfortable API für Abfragen, Insertionen, Updates und Löschungen. Diesel verwendet das Rust-Typensystem, um Compile-Time-Prüfungen durchzuführen, was zu robusterem Abfragen und einer starken Typisierung der Datenbankzugriffe führt.

Generell lässt sich die API in vier separate Module unterteilen, die jeweils bestimmte Endpunkte für verschiedene Aufgabe an derselben Ressource anbieten. Diese Module sind:

- Player: Hierüber lassen sich die Informationen bezüglich der Spieler abfragen, bearbeiten und synchronisieren.
- Building: Hierüber lassen sich die Informationen bezüglich aller Buildings im Spiel abfragen und bearbeiten.
- Upgrade: Hierüber lassen sich die Informationen bezüglich aller Upgrades im Spiel abfragen und bearbeiten.
- Purchase: Mithilfe dieses Moduls wird vor allem die Synchronisation zwischen Backend und Frontend bei der Produktionsrate und der Geldmenge vorgenommen.

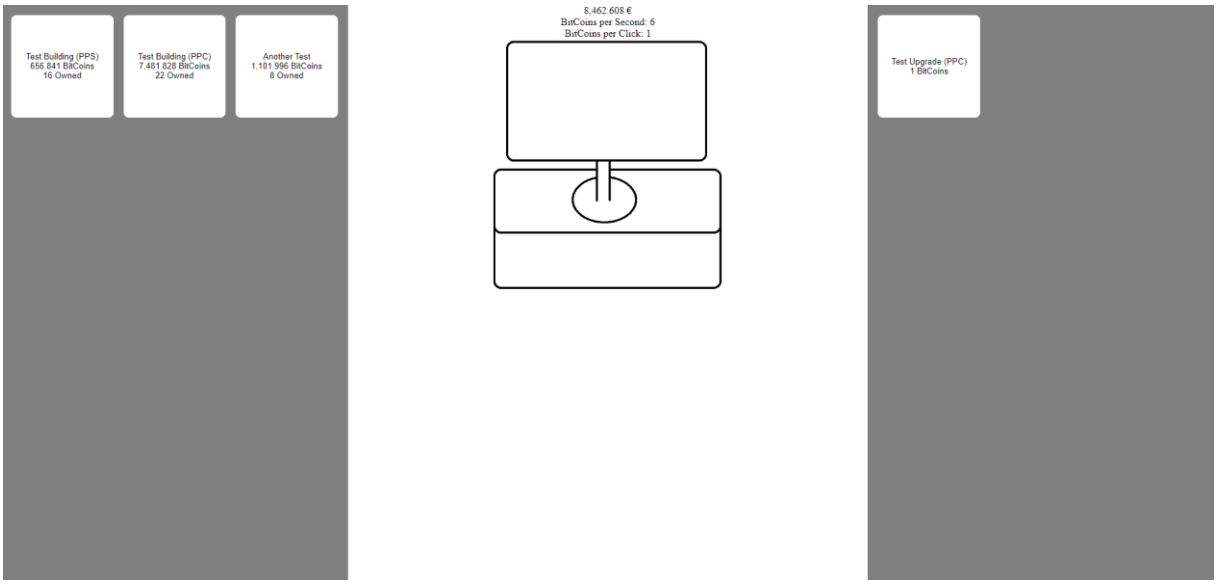
Weitere Informationen lassen sich aus dem GitHub-Repository entnehmen.

Umsetzung des Frontends

Das Frontend wurde mit Angular umgesetzt. Wir haben uns für Angular entschieden, da dies jedem der Projektbeteiligten vertraut ist und daher weniger Zeit in der Entwicklung benötigt. Dies ist vor allem deshalb relevant, weil Rust jedem Projektbeteiligten komplett unbekannt war und daher aus Lernzwecken Fokus dieses Projektes ist – ein zweites unbekanntes Tech-Stack würde die Entwicklung somit erheblich verlangsamen.

Das Scaffolding der Angular Applikation wurde hierbei mit NX umgesetzt. Typischerweise wird NX im Zusammenhang von Monorepos verwendet, jedoch bietet es neben Generierungsskripten von Boilerplate-Code auch die Funktionalität des Zwischenspeicherns von CI/CD Ergebnissen, was viele der unterschiedlichen Entwicklungswerkzeuge

beschleunigt.



Wie bereits dem Nutzerhandbuch zu entnehmen ist, ist das Frontend hauptsächlich aus drei Arealen aufgebaut. Links findet man die Gebäude, welche die Produktion pro Klick oder auch die Produktion pro Sekunde um einen additiven Faktor verbessern können. Diese Gebäude können mehrfach erworben werden, jedoch steigern sich die Kosten pro Kauf anhand folgender Formel:

$$Kostenmultiplikator^{Anzahl\ der\ bisherigen\ Käufe} * Basiskosten\ des\ Gebäudes$$

Rechts im Clicker-Frontend befinden sich die Upgrades. Upgrades sind lediglich einmalig erwerbbar, verbessern jedoch entweder die Produktion eines Gebäudes oder gar die Gesamtproduktion multiplikativ.

Die Anzeige in der Mitte zeigt zunächst die wichtigsten Werte an, wie die Anzahl an Währung, die Produktion pro Klick, sowie die Produktion pro Sekunde. Außerdem beinhaltet sie den „Computer“, welcher Klickbar ist, und dadurch die Währung erhöht.

Mechanismen gegen ungewünschte Einkommensmanipulationen

Um die Integrität der Daten zu schützen, wurden mehrfach Vorkehrungen gegen beabsichtigte oder unbeabsichtigte Manipulationen implementiert.

Zunächst ist vorgesehen, dass das Frontend regelmäßig mit dem Server eine Synchronisierung ausführt. Hierbei wird zunächst der Zeitstempel vom Server abgefragt, zu welchem Zeitpunkt zuletzt der Währungsstand berechnet wurde. Abhängig davon bestimmt nun das Frontend basierend auf der Produktion Zwischenwerte, mit welchem der Spieler einsehen kann, wie viel Währung er gerade besitzt. Sollte er nun ein Gebäude oder Upgrade erwerben, wird eine dieser Synchronisationen durchgeführt. Dafür übermittelt das Frontend einmal den Zeitstempel, den es als letzten Zeitstempel empfangen hat, den aktuellen Zeitstempel, sowie die Anzahl an Klicks, die in der Zwischenzeit getätigt wurden.

Abhängig davon wird nun im Backend folgendes überprüft:

1) Liegt der mitgelieferte „aktuelle Zeitstempel“ in der Vergangenheit? Falls nicht, sind die Daten ungültig.
2) Liegt die Anzahl der Klicks über einem gewissen Wert pro Sekunde? Falls ja, sind die Daten ebenfalls ungültig.
3) Ist der Ursprungszeitstempel nicht mit dem Identisch, den das Backend noch gespeichert hat, werden die Daten ebenfalls verworfen.

Sollte keine dieser Bedingungen fehlschlagen, wird im Backend einmal die Währung anhand der Produktion nachgerechnet. Daraufhin wird dann geprüft, ob genug Währung für das Kaufen des Gebäudes oder der Währung vorliegt und die Produktion bei erfolgreichem Erwerb neu berechnet.

Berechnung der Produktion

Um die Produktion zu berechnen, wird zunächst ausgewertet, wie viel jedes Gebäude produziert. Dafür wird die Basisproduktion eines Gebäudes mit der Anzahl multipliziert. Dann wird jedes Upgrade betrachtet, das der Spieler besitzt. Upgrades haben hierbei drei mögliche Faktoren, welche in einer Liste gespeichert werden. Wenn mehrere Upgrades denselben Faktor betreffen, werden diese additiv zusammengerechnet. Nach Betrachtung aller Upgrades werden diese Faktoren dann multiplikativ verrechnet. Zum Schluss werden „Globale“ Upgrades mit der zusammengerechneten Produktion multipliziert.

Testing und Polishing

Aus Zeit- und Komplexitätsgründen wurde sich gegen automatische Tests entschieden. Diese bieten zwar einen enormen Mehrwert für die Sicherstellung der Codequalität und Stabilität, jedoch sind diese gerade in einer schnell wandelnden Umgebung wie einem Prototypen in einer fremden Programmiersprache besonders aufwändig. Durch jede grundlegende Architekturentscheidung, welche gerade bei den ersten Lernprozessen häufig vorkommen, müssten viele der Tests neu geschrieben werden. Daher wurde lediglich White-Box-Testing durchgeführt, um möglichst viele Fehlerfälle abzufangen.

Reflexion der Projektarbeit

In der Planungsphase haben wir, nach eigener Ansicht, einige Fragen hinsichtlich des Zeitbudgets für das Projekt erheblich zu weit gedacht. Dennoch haben wir grundlegend gute und durchdachte, berechtigte Entscheidungen getroffen, sodass der weiteren Entwicklungsarbeit hierdurch keine Probleme bereitet wurden.

Wie erwartet, erbrachte die Einarbeitung in Rust als neues Tech-Stack sowie einige dazugehörige Toolchains zunächst einige Hindernisse, diese wurden jedoch ausreichend schnell überwunden, sodass das Backend und damit die Grundlage unseres Clickers begonnen werden konnten.

Den letzten Feinschliff hat das Backend zwar leider noch nicht erhalten, jedoch ist es in einer stabilen, übersichtlichen und nutzbaren Verfassung gut dazu geeignet, das Spiel grundlegend kennenzulernen und auch zu spielen.

Autoren: Danny, Mike, Julian

Auch die Dokumentation ist zwar nicht an jeder Stelle perfekt oder gar “idiotensicher”, bietet jedoch eine solide Basis zum Einarbeiten in die Spieler- und Entwickler-Perspektive bezüglich unseres Clickers.