# Datenverwaltung

Zum persistenten speichern und verwalten der Daten haben wir uns für eine SQLite Datenbank entschieden. ­

## Datenbank Manager

Als Schnittstelle für die Kommunikation mit der Datenbank wurde ein Datenbank Manager implementiert.

Dieser stellt zum einen allgemeine Funktionen für die jeweiligen Klassen zur Verfügung, als auch die Methoden zur initialen Tabellen Erstellung. Die vorher manuell angelegte Datenbank liegt bereits im Projektordner. Der Datenbankname muss beim Anlegen eines DBManager-Objekts an die Funktion „initWithDatabaseFilename“ übergeben werden. In dieser Funktion wird der Verzeichnispfad zu der Property „documentsDirectory“ hinzugefügt und die Datenbank-Datei falls notwendig in das Dokumenten-Verzeichnis kopiert.

Darüber hinaus gibt es noch die Funktion „runQuery“, die im Folgenden Erläutert wird, da sie ein Stück weit das Herzstück des DBHelper’s ist. Diese nimmt einem alle Schritte die zum Ausführen einer Datenbank-Anfrage nötig sind. Zum einen erstellt sie ein Ergebnis-Array und ein Spaltennamen-Array. Dieses ermöglicht es einen später statt einem Zugriff per Index einen Zugriff per Spaltennamen.

Bei den Anfragen wird eine allgemeine Unterscheidung zwischen ausführbaren Anfragen und nicht ausführbaren gemacht, also ob es lediglich eine Datenabfrage ist, oder ob etwas in der Datenbank ausgeführt wird. Die nötige Information dafür bekommt die „runQuery“-Funktion dadurch, dass wir sie meistens indirekt über die Funktionen „loadDataFromDB“ und „executeQuery“ aufrufen.

Diese geben eine BOOL Variable an die Funktion, in der steht, ob die Anfrage ausführbar ist oder nicht. Über diesen Wert wird in der „runQuery“-Funktion nun eine Fallunterscheidung gemacht.

Der Rest der Funktion besteht nun nur noch aus dem Auslesen der Daten und dem Befüllen der beiden angelegten Arrays.

## Inhalt und Klassen

Die Datenbank besteht im Allgemeinen aus drei Tabellen, die in dieser Version des Projektes in keiner Beziehung miteinander stehen. Gegen diese haben wir uns entschieden, weil wir in unserer App grundsätzlich nicht erlauben wollen, dass auf einem Gerät mehrere Piraten gleichzeitig angemeldet sind und ohne diese Funktionalität wären die Beziehungen nur ein Overhead (bei jeder Anfrage auf eine der Detailtabellen müsste immer ein Join über den Piraten eingebaut werden).

In den folgenden Abschnitten werden unsere drei Tabellen zusammen mit deren Funktionalitäten vorgestellt.

### Piraten

In der Piraten-Tabelle werden sämtliche Informationen zum aktuellen Piraten verwaltet.

Der Spieler wählt zu Beginn eines neuen Spiels einen Namen aus, zusätzlich bekommt jeder neue Pirat eine bestimmte Anzahl an Leben, ein Startlevel und einen Standard Alkoholpegel zugewiesen (wo festgelegt wird welchen Wert diese Werte haben sollen und wie es umgesetzt wird siehe Kapitel TypeDef). Das Feld „geschaffteBeduerfnisse“ gibt an wie viele Bedürfnisse der Pirat seit seinem Erstellen erfolgreich erledigt hat. Dieses Feld ist auch der Richtwert für einen Level-Aufstieg.

Nach jedem geschafften Bedürfnis wird innerhalb des Piraten-Objekts die Methode „checkLevelUp“ aufgerufen, die nach ebenfalls in der TypeDef abgelegten Werten prüft, ob genug Bedürfnisse für einen Level-Aufstieg erfüllt sind. Selbiges geschieht für den Alkohol-Pegel, wo entschieden wird, ob der Pirat betrunken ist oder nicht.

Die Piratenklasse hat neben diesen Funktionen auch noch die einzelnen Funktionen, seine jeweiligen Attribute zu erhöhen oder zurück zu setzen.

Eine zusätzliche Eigenschaft dieser Klasse ist, dass der Pirat bei Spiel Start seine Klassen-Attribute mittels der „loadData“ Funktion aus der Datenbank übernimmt, wodurch wir zur Laufzeit nur eine lokale Kopie des Piraten haben und uns so ein paar auslesende Zugriffe auf die Datenbank ersparen können. Nach jeder Änderung wird die Datenbank parallel zur lokalen Instanz geupdatet um bei einem möglichen Spielabsturz nicht die Daten des Piraten-Objekts zu verlieren.

Zusätzlich gibt es noch die Funktion „saveData“, die im Allgemeinen zu Testzwecken da war, falls man bei der Entwicklung gerne ein Attribut ändern wollte, wofür es keine Funktion gab und man somit das Objekt direkt verändert hat. Mit der „saveData“-Funktion konnte man somit das komplette Objekt in die Datenbank einspeichern.

### Lager

In der Lager-Tabelle werden die einzelnen Waren verwaltet.

Die Waren an sich sind fix und werden bei einem neuen Spiel mit Hilfe der TypeDef-Datei in die Datenbank geschrieben. Genauso hat jeder neue Spieler auch eine gewisse Startanzahl. Das Geld wird hier ebenfalls als Ware behandelt und verwaltet. Hier sind die Kosten „1“ und die Anzahl bestimmt den Kontostand.

Das Lager-Objekt wird genauso wie das Piraten-Objekt mittels der „loadData“-Methode gefüllt. Innerhalb der Klasse gibt es noch die Lagerveraltungsfunktionen „buy“ um ein Item im Shop zu kaufen, „give“ um dem Spieler ohne weiteres ein Item zu geben (wird für die Raubzug-Option gebraucht), „sell“ um ein Objekt aus seinem Lager zu verkaufen (Verkaufspreis berechnet sich aus der Hälfte des Einkaufspreises) und „useItem“ um ein Item zu benutzen und dadurch Wünsche des Piraten zu erfüllen. Nach jedem Aufruf einer solchen Methode, die nach Überprüfung der erforderlichen Mittel auch erfolgreich ist, wird die „update“-Methode aufgerufen, die die Datenbank direkt aktualisiert.

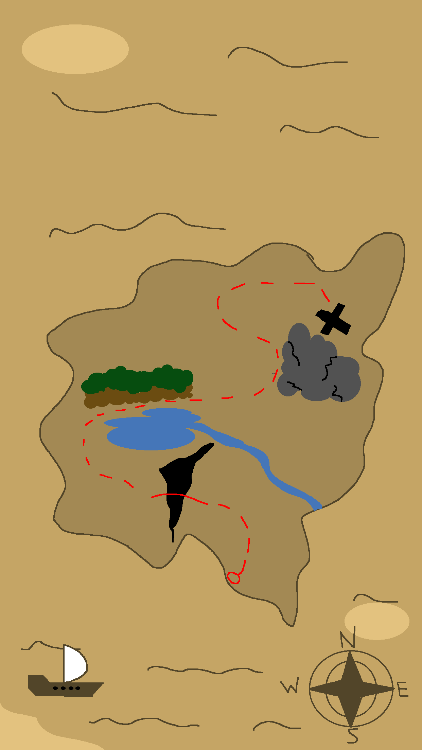
Auch das Lager hat die „saveData“-Funktion. Diese hat den gleichen Zweck wie im Piraten.

### Aktuelle Bedürfnisse



# Minispiel

## Schatzsuche

Im diesem Minispiel geht es um eine Art Schatzsuche. Der Spieler muss hier mit dem Handy 100 Meter laufen um den Schatz in Höhe von 10 Münzen zu erhalten.

Technisch wird das ganze mit dem GPS-Sensor umgesetzt, da dies über den Emulator mit der „StreetRun“-Funktion simulierbar ist. Sobald man auf „Start“ drückt sieht man auf dem Display wie viele Meter man noch laufen muss. Die App berechnet nach jeder neuen Standort-Aktualisierung die Distanz in Metern zwischen dem letzten Standort und dem neuen und verrechnet die neuen Meter mit den bisher zurückgelegten. Sobald man die 100 Meter zurückgelegt hat, bekommt man die 10 Münzen gutgeschrieben und man hat die Möglichkeit die Schatzsuche erneut zu starten.

Ebenfalls ist es möglich die App zu minimieren, denn der LocationManager merkt sich alle Standortveränderungen. Genauso gut ist es bei aktiviertem GPS Sensor auch möglich zu einem späteren Zeitpunkt die Suche zu starten (wenn sie nach App-Start bereits einmal gestartet wurde) und sich die bis zu Neustart zurückgelegten Meter einmalig auf die neue Schatzsuche gut schreiben zu lassen.

Beispiel: Wenn man zwischen den Schatzsuchen 30 Meter gelaufen ist und dann die neue Schatzsuche startet, muss man nur noch 70 Meter zurücklegen.

*Tipp: Wenn man auf den „Zurück“-Button drückt, um sich zum Beispiel um seinen Piraten zu kümmern, werden die bisher geschafften Meter zurückgesetzt. Deswegen ist es zu empfehlen zu laufen, bevor man „Start“ drückt um später mit den nebenbei gelaufenen Metern direkt in eine neue Suche zu starten. Das hat die Ursache, dass wir für die Schatzsuche keine extra persistente Speicherung erstellen wollten (um die bisher zurückgelegten Meter zu speichern). Genauso startet man deshalb auch nach kompletten App-Neustart wieder bei 100 verbleibende Metern.*

# TypeDef

Die „TypeDef.h“-Datei haben wir angelegt, um dort verschiedene Konstanten anzulegen.

Beispielsweise werden dort die Werte gespeichert, die bei Initialisieren eines neuen Spielstandes in die Datenbank eingetragen werden. Auch die Array-Indizes für das „supllie“-Array im Storage-Objekt werden hier angelegt. Dies ermöglicht es, nicht jedes Mal das Array durchlaufen zu müssen und nach dem richtigen Eintrag zu filtern, sondern direkt über die Konstante darauf zu zugreifen. Dadurch, dass auch die Datenbank nach diesem Schema erstellt wird, werden die Indizes immer stimmen.

Auch sämtliche Limits, wie Levelaufstieg-Grenzen, werden hier festgelegt.