Inhaltsverzeichnis

1 Vorwort 2
2 Das Projekt 3
3 Vorarbeiten 4
3.1 Python 4
3.2 Die IDE 6
3.3 SQLiteStudio 9
3.4 Die beiden Browser 10
4 Projekt „Schiffeversenken" 11
4.1 Der Funktionsumfang 11
4.2 Die Planung der Umsetzung 12
4.2.1 Client-Server-Architektur 12
4.2.2 Das Framework 12
4.2.3 Die Datenbank 13
4.2.4 Weitere Sprachen 13
4.2.5 Noch etwas? 13
4.3 Das Coden 14
4.3.1 Arbeiten mit Flask 14
4.3.2 SQLite 23
4.3.3 Der Spielstart 32
4.3.4 Der Spielcode 37
4.3.5 Die Spielfelder 50
4.3.6 Setzen der Schiffe 59
4.3.7 Das Spielen 70
4.4 Die Veröffentlichung 78
5 Abschluss 87

1 Vorwort

Dieses Projekt hat jetzt länger gedauert, mein Neffe und ich haben uns in den Kopf gesetzt, dass wir das Spiel „Schiffeversenken“ nachbauen wollen. Dafür habe ich viele Sachen ausprobieren müssen, um den richtigen Ansatz zu finden. Aber es hat sich gelohnt, das Spiel ist online, ohne dass wir für das Hosten der Anwendung bezahlt hätten.

Was gibt es also diesmal zu entdecken?

Bestandteil der Doku ist die Programmierung des Source-Codes in Python, HTML, CSS und JavaScript, als Datenbanksystem habe ich mich für SQLite entschieden. Viele der Hürden konnten auch dank des Einsatzes von ChatGPT genommen werden.

Die Entwicklung erfolgt auf dem eigenen Rechner, am Schluss zeige ich aber noch, wie man das Spiel ohne Kosten im Internet hosten kann. Über papasbattleship.pythonanywhere.com kommt Ihr auf die in Kapitel 4.4 beschriebene Version. Der Name ist etwas sperrig, aber es funktioniert.

Ich werde versuchen, Euch Schritt für Schritt auf unsere Reise mitzunehmen, damit sollte dem Nachbau nichts im Wege stehen und am Ende habt Ihr Eure eigene Anwendung. Das kann dann als Basis für weitere Online-Spiele dienen.

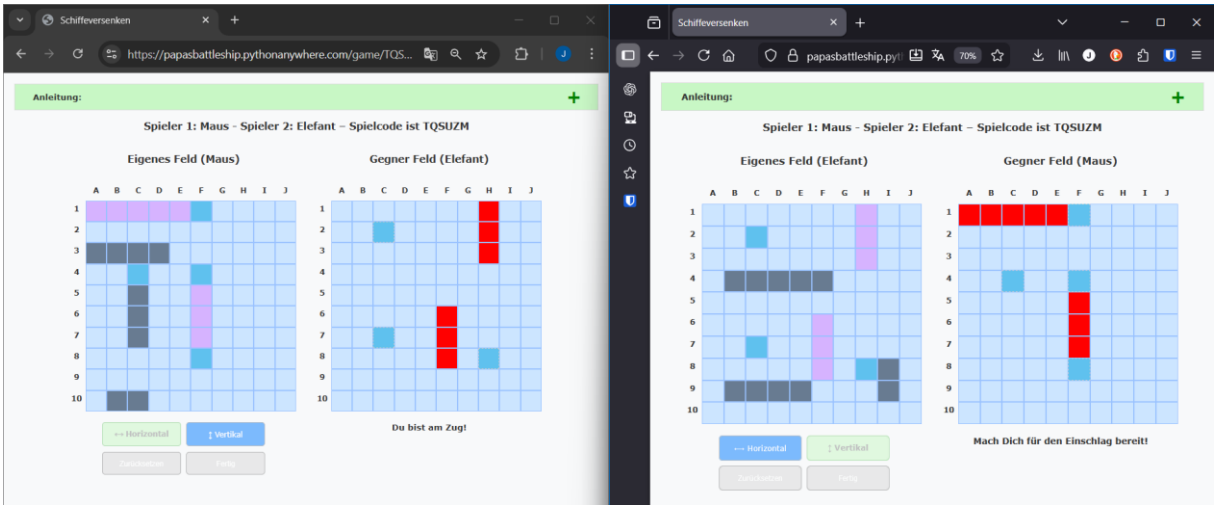
Wie immer an dieser Stelle der Aufruf, meldet Euch gerne bei mir unter papa@papa-programmiert.de, ich bin für Kritik und Anregungen offen und immer zu haben.

Und jetzt viel Spaß!

2 Das Projekt

Wie im Vorwort beschrieben, geht es in diesem Projekt um das Spiel „Schiffeversenken“.

Um Euch einen ersten Eindruck zu geben, hier das fertige Ergebnis: Ich habe gegen mich selber gespielt, rechts ist eine Firefox-Sitzung zu sehen, links eine Chrome-Sitzung.



Das Spielprinzip ist klar, oder? 2 Spieler verteilen 5 Schiffe auf ihrem Spielfeld, sodass sie sich nicht berühren oder überlappen. Wenn beide Spieler alle Schiffe gesetzt haben, startet das Spiel mit der zufälligen Auswahl eines der beiden Spieler als Startspieler. Dieser klickt nun im gegnerischen Feld auf ein beliebiges Kästchen. Gehört dieses Kästchen zu einem Teil eines gegnerischen Schiffs, wird der Schuss als „Treffer“ gewertet und der Spieler darf noch einmal klicken. Liegt dort kein Teil eines gegnerischen Schiffes, wird der Treffer als „Wasser“ gewertet und der gegnerische Spieler ist am Zug.

Soweit so einfach. Der Teufel steckt im Detail, das werden wir später sehen.

Dann also auf, lasst uns gleich zu den Vorarbeiten kommen.

3 Vorarbeiten

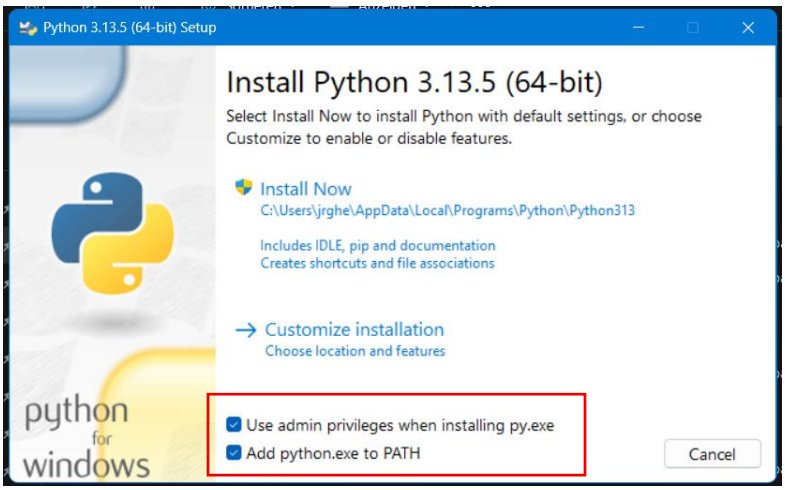
Für dieses Projekt sind die Vorarbeiten etwas umfangreicher.

Wir brauchen **Python**, eine **Entwicklungsumgebung**, einen **Datenbankeditor** und **zwei Browser**.

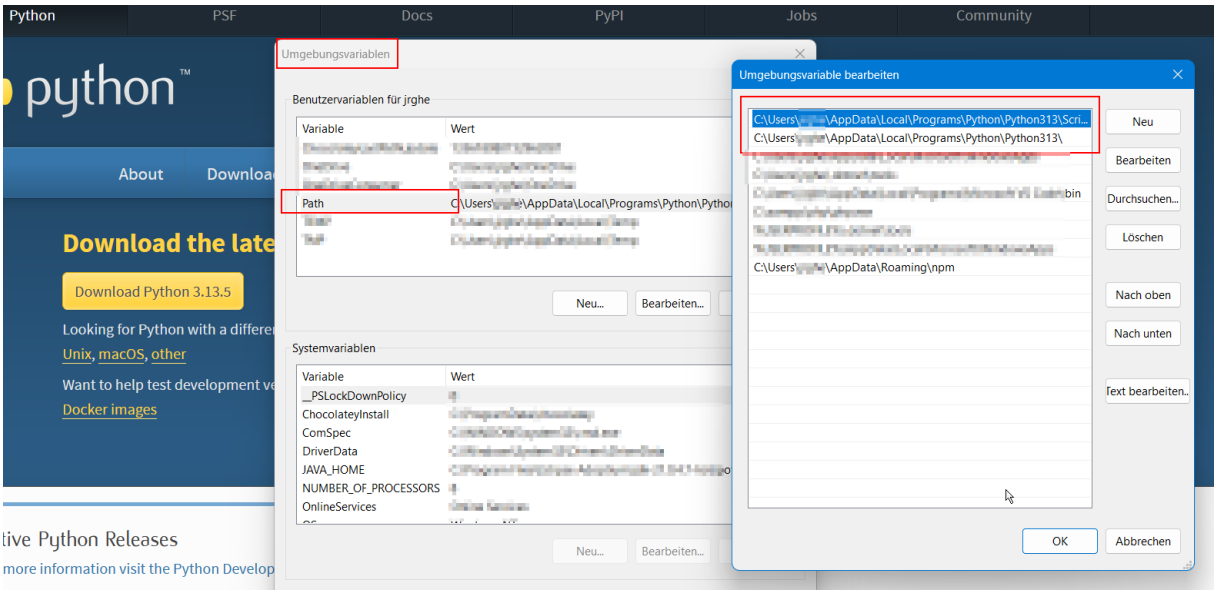
3.1 Python

Um in Python entwickeln zu können, müssen wir Python auf unserem Rechner installieren. Die neueste Version von Python gibt es unter www.python.org/downloads/. Zum Zeitpunkt meiner Entwicklung war das 3.13.5. Der Download umfasst etwa 27 MB.

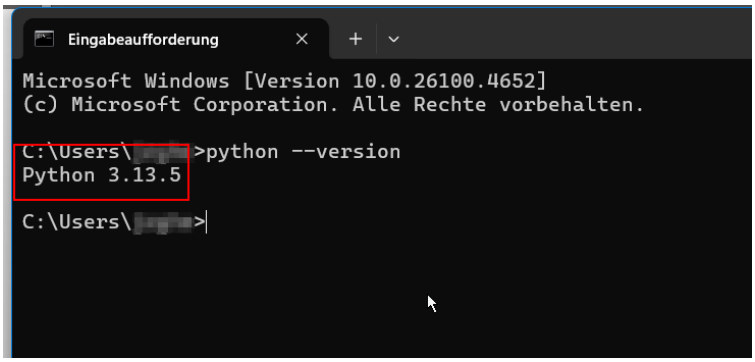
Die Installation ist gut geführt, Ihr werdet keine Probleme damit haben. Wichtig am Anfang, die beiden Häcken setzen, damit die Umgebungsvariablen im PATH gesetzt werden:



Das sollte dann so aussehen:



Falls Ihr damit Probleme haben solltet, im Internet gibt es dafür jede Menge Hilfe. Um den Erfolg der Installation zu testen, könnt Ihr eine Terminalfenster starten (in Windows unten links in die Suchleiste „`cmd`“ eingeben, das führt Euch zur „Eingabeaufforderung“) und `python -version` eingeben, dann sollte die korrekte Version angezeigt werden:

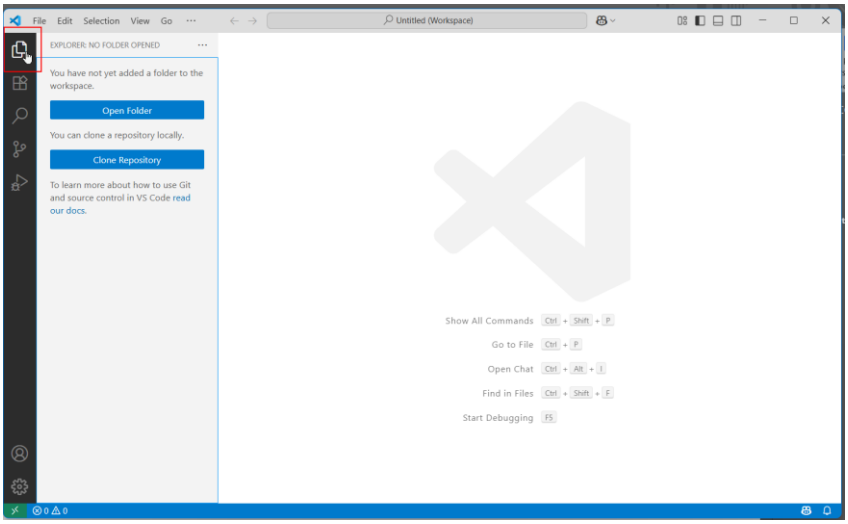


3.2 Die IDE

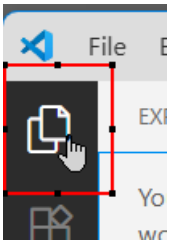
Da die Entwicklung sowohl Python als auch HTML und CSS umfasst, habe ich mich für den Umstieg von Thonny zu Visual Studio Code („VSCode“) als Entwicklungsumgebung entschieden. Die IDE ist kostenlos, von Microsoft entwickelt und hat einen großen Funktionsumfang. Sie läuft auch unter Linux, ist somit beispielsweise auch für Ubuntu-Nutzer nutzbar.

Das ist aber natürlich nicht die einzige Möglichkeit, wer eine Alternative sucht, könnte sich „Eclipse“ anschauen, auch kostenlos und sogar Open Source. Ebenfalls kostenlos aber relativ unbekannt ist die „Netbeans IDE“ von der Apache Foundation. Jetbrains hat gleich mehrere IDEs im Angebot, für jede Programmiersprache eine eigene, für Python wäre das „PyCharm“. Es gibt immer eine kostenlose Version, allerdings ist der Funktionsumfang bei diesen Versionen eingeschränkt. Für die Nutzung aller Funktionen gibt es eine kostenpflichtige Version. Aus er Schweiz kommt die IDE „Spyder“.

VSCode gibt es hier (https://code.visualstudio.com/Download). Download und Installation sind relativ einfach, das schafft Ihr auch ohne meine Hilfe. Nach dem Start sollte es etwa so bei Euch aussehen:

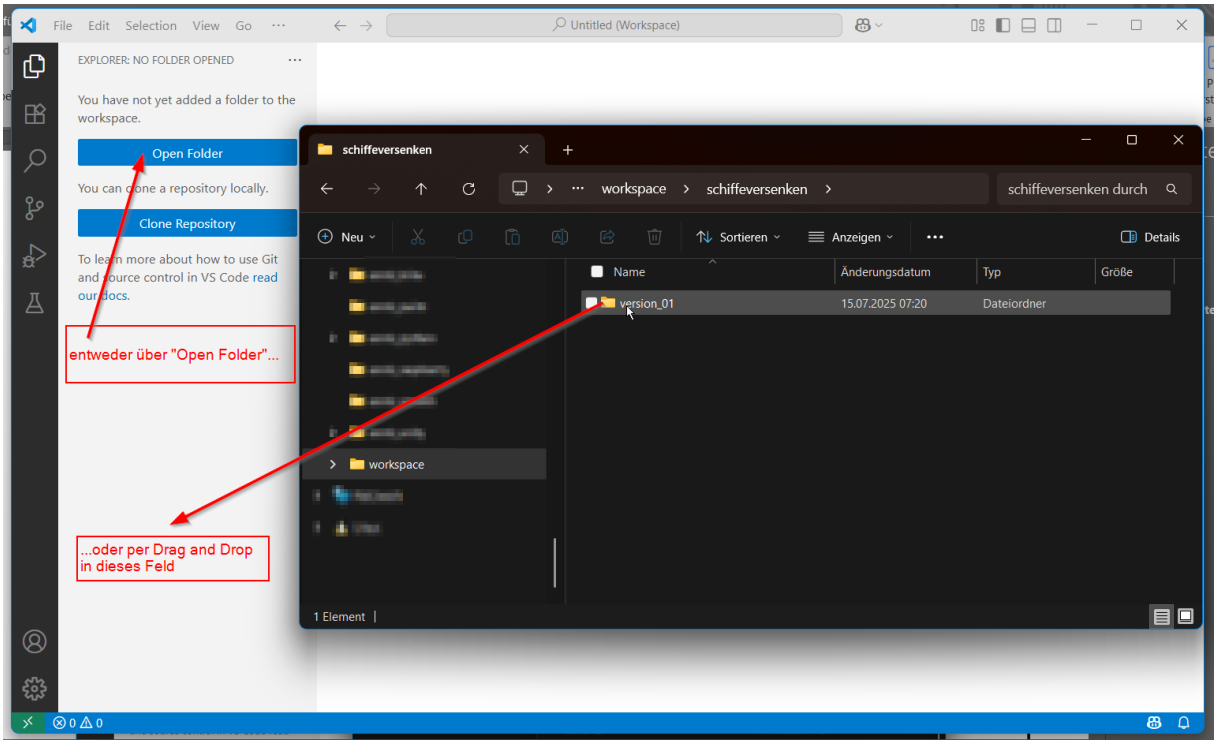


Die Ansicht mit dem Ordner könnt Ihr mit Klick auf das Datei-Symbol oben links ein- und ausschalten:

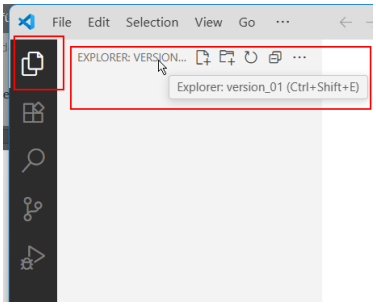


Tipp: Nach meiner Erfahrung ist es sinnvoll, einen Ordner zu haben, in dem alle Sourcen enthalten sind, so etwas wie C:/workspace. Dann folgen Ordner für die einzelenen Projekte also C:/workspace/schiffeversenken, und C:/workspace/taschenrechner. Darunter habe ich immer noch die einzelnen Versionen, da ich mir gerne vor entscheidenenden Änderungen eine Sicherung mache. Bei mir sieht die Ordnerstruktur so aus: C:/workspace/schiffeversenken/version_01. Darunter finde ich dann alle Sourcen zu diesem einen Stand.

Egal wie Ihr Euch organisiert, in VSCode kann man den ganzen Ordner entweder über „Open Folder“ selber auswählen, oder ihn per Drag&Drop in den Dateiansichtsfenster links ziehen:

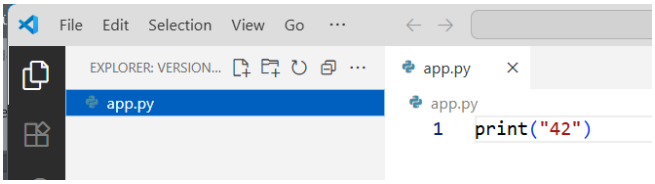


Wenn Ihr den Ordner angelegt und in VSCode engezeigt habt, sollte das bei Euch so aussehen:

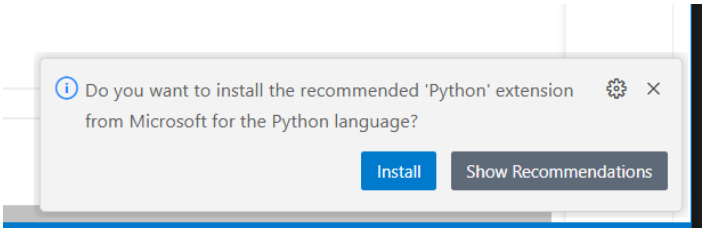


Um die Installation von VSCode zu prüfen, erstellen wir eine neue Datei mit Namen `app.py` über File/New File oder im Explorer, da wir den Ordner eingebunden haben, wird auch VSCode aktualisiert.

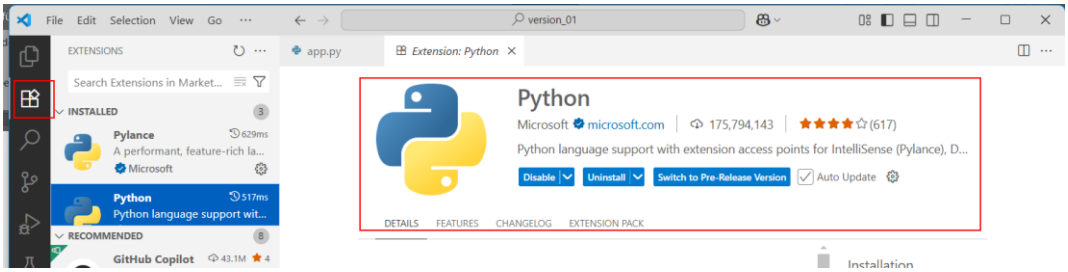
In der app.py geben wir den code `print(„42“)` ein:



VSCode sollte Euch fragen, ob Ihr die Erweiterung für Python installieren wollt, das wollt Ihr natürlich und klickt „Install“:

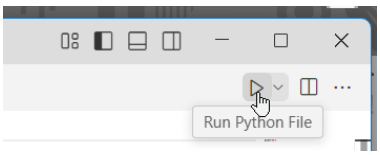


Damit sollten die ersten beiden Erweiterungen (Pylance und Python) installiert werden:

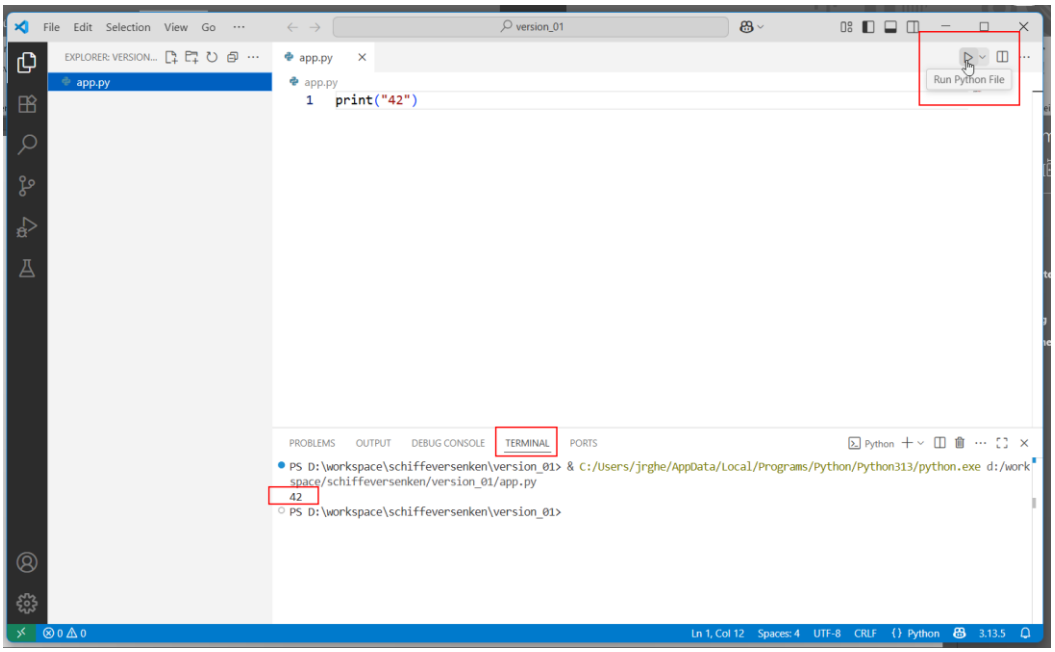


Erweiterungen sind über die 4 Quadrate links aufruf- und installierbar.

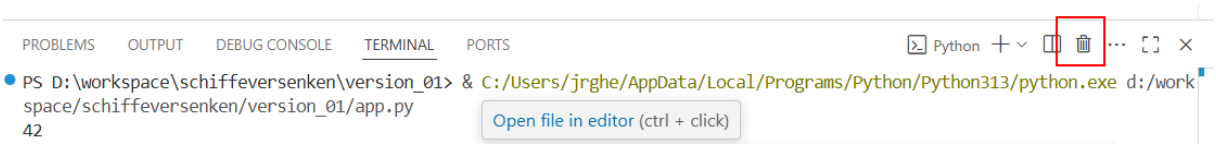
Mit den beiden neuen Erweiterungen sollte oben rechts auch ein Dreieck auftauschen, mit dem Ihr die `app.py` starten könnt:



Es öffnet sich dann ein Terminal-Fenster unten, in dem das Ergebnis angezeigt wird:



Um das Terminalfenster zu schließen, klickt auf das Delete-Symbol rechts in der Menu-Leiste des Terminals:



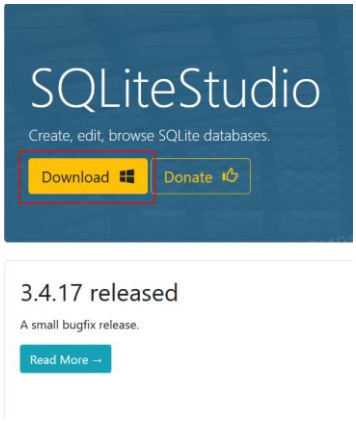
Damit sind wir mit der IDE startklart, weitere Infos finden sich dann im Programmierteil.

3.3 SQLiteStudio

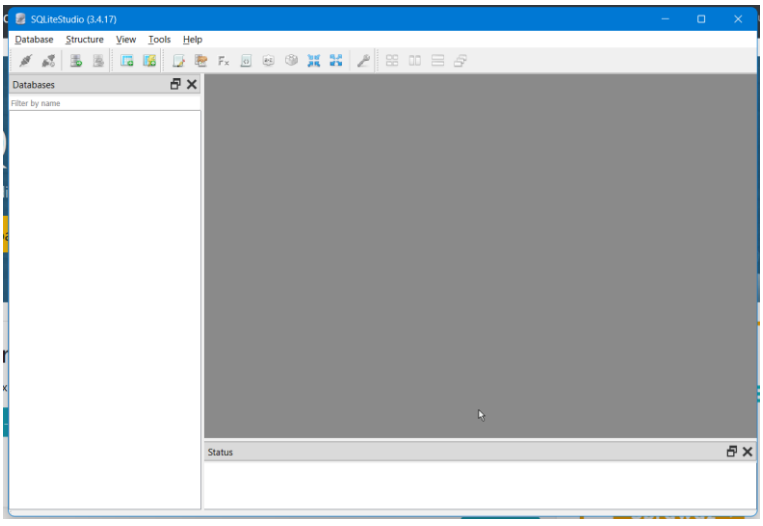
Nach einiger Suche habe ich mich als Datenbanksystem für `SQLite` entschieden. Ich hätte auch `MariaDB` oder `MySQL` nehmen können, da wir `Flask` einsetzen (dazu später mehr), habe ich viele Foren und Tutorials gesehen, in denen eine SQLite-Datenbank eingebunden war.

Um eine Datenbank verwalten zu können, brauchen wir ein Werkzeug, mit dem wir die Datenbank und Tabellen darin anlegen, ändern und löschen können. Für SQLite haben Entwickler aus Polen das hübsche SQLiteStudio geschrieben.

Der Download ist über die Seite https://sqlitestudio.pl/?act=download%E3%80%82SQLiteStudio erreichbar.



Da ich Windows nutze, lade ich mir die `exe` herunter und installiere sie. Da wir noch keine Tabellen haben, sehen wir nach Start nur ein leeres Fenster:



3.4 Die beiden Browser

In der Einleitung zu Kapitel 3 habe ich geschrieben, dass wir 2 Browser brauchen. Warum das? Zum einen, weil wir die Entwicklung auf unserem eigenen Rechner machen, der Rechner ist also der Server. Wenn wir mit nur einem Browser 2 Spieler simulieren würden, könnten wir die Struktur der Client-Server-Anwendung (dazu später mehr) eventuell falsch umsetzen.

Zum anderen ist es aber auch gut, dass wir HTML-Darstellungen immer in möglichst vielen unterschiedlichen Browsern testen, schließlich soll unsere Anwendung mit jedem Browser klarkommen. Bei den unterschiedlichen Browsern ist die Darstellung auch immer etwas anders, das sollten wir uns zumindest mal angesehen haben, bevor wir das online stellen.

Bei mir kommen aktuell `Firefox` und `Chrome` zum Einsatz. Testen werde ich aber auch `Edge` und `Opera`.

Das war es mit den Vorbereitungen, diesmal etwas mehr, da wir auch eine neue IDE installiert haben.

4 Projekt „Schiffeversenken"

Wie auch in den anderen Projekten schon praktiziert, machen wir uns zuerst Gedanken über den Funktionsumfang, also die Frage nach dem ‚**Was**'. Das strukturiert das Projekt und wir haben eine klarere Vorstellung, als wenn wir direkt „draufloshacken" würden.

Dann gehen wir das ‚**Wie**' an. Fest steht ja schon, dass wir das mit Python lösen wollen, den Rest schauen wir uns dann im Kapitel „Planung der Umsetzung" an.

Im dritten Teil dann kommt die Programmierung dran.

4.1 Der Funktionsumfang

Das Spiel Schiffeversenken kennt Ihr vermutlich alle, oben habe ich das ja auch noch einmal skizziert. Damit wir eine gemeinsame Basis haben, legen wir in diesem Kapitel fest, was das Spiel können soll, und was eventuell in einer späteren Ausbaustufe kommen soll oder kann.

Wir wollen ein Spiel entwickeln, bei dem 2 Spieler über das Internet gegeneinander das Spiel Schiffeversenken spielen können.

Die Spielfelder sind 10 mal 10 Kästchen groß, es kommen je Spieler 5 Schiffe zum Einsatz. Ein Schiff mit 5 Kästchen Länge, eines mit 4 Kästchen, 2 Schiffe mit 3 Kästchen und eines mit 2 Kästchen.

Die beiden Spieler sollen sowohl ihr eigenes Spielfeld, als auch das des Gegners sehen können, wobei auf dem eigenen Spielfeld die Schiffe sichtbar sind, auf dem des Gegners nur die Schüsse markiert sind. Die Schüsse des Gegners werden auf dem eigenen Spielfeld angezeigt.

Der Spielablauf ist wie folgt, beide Spieler melden sich am System an und verteilen ihre Schiffe auf dem eigenen Spielfeld. Wenn das geschehen ist, ist das Spiel bereit, Änderungen an der Position der Schiffe dürfen nicht mehr gemacht werden. Per Zufall wird ein Spieler als Starter ermittelt. Er beginnt das Spiel, indem er seinen ersten Schuss abgibt. Der Schuss wird auf dem Spielfeld des Gegners markiert.

Sollte der Schuss ein „Treffer" sein, also auf ein Feld getroffen sein, auf dem ein gegnerisches Schiff platziert ist, darf der Spieler erneut schießen. Ging der Schuss ins Wasser, ist der andere Spieler an der Reihe. Auf dem gegnerischen Spielfeld müssen „Treffer" und „Wasser" farblich unterschiedlich gestaltet sein.

Das Spiel wechselt solange hin und her, bis einer der beiden Spieler zuerst alle Schiffe des Gegners getroffen hat.

Am Ende sollen beide Spieler noch einmal ihr eigenes Spielfeld und das des Gegners mit der Position der Schiffe, aller Treffer und Fehlschüsse angezeigt bekommen.

In einer späteren Ausbaustufe soll es möglich sein, dem Spieler eine Statistik anzuzeigen, die die Anzahl seiner gespielten Spiele sowie deren Ergebnisse ausweist.

Damit lässt sich arbeiten, oder? Dann gleich weiter zum ‚**Wie**'.

4.2 Die Planung der Umsetzung

Hier sind jetzt wir als kreative Techis gefragt.

Nach intensiven Internetrecherchen haben wir folgende Informationen zusammengetragen:

- Um Dinge über das Internet bereitzustellen, brauchen wir eine **Client-Server-Architektur**
- Wenn wir Web-Entwicklung mit Python machen wollen, sollten wir ein **Framework** wie Flask, Django oder andere nutzen.
- Serverseitig sollten wir eine **Datenbank** im Einsatz haben
- Python kommt in der Server-Programmierung zum Einsatz
- HTML kommt auf der Client-Seite zum Einsatz
- Die Aufbereitung des HTML-Codes erfolgt mit CSS
- Für die Logik zwischen Client und Server sollte JavaScript genutzt werden

Ziemlich viele Infos, die wir irgendwie noch einsortieren müssen. Falls Euch das alles bekannt ist, überfliegt oder überspringt die nächsten Kapitel gerne.

4.2.1 Client-Server-Architektur

Es gibt viele gute Beiträge im Internet die erklären, was eine Client-Server-Architektur überhaupt ist. Meine Herleitung resultiert aus dem Beispiel einer Internetrecherche:

Wir geben den Suchtext in unserem Browser ein, drücken dann Eingabe und erhalten die Trefferliste als Suchergebnis zurückgeliefert. Unser Browser ist dabei also der **Client**, der die Anfrage stellt, aber keine eigenen Informationen hält.

Die Frage geht dann an den Betreiber der von uns ausgewählten Suchmaschine und landet dort auf deren **Servern**. Dort kümmert man sich um die Beantwortung der Frage und schickt das Ergebnis an den Aufrufer zurück. Meist ist es nur das erste Paket von ganz vielen Treffern, der Server teilt dem Aufrufer dann mit, dass es weitere Infos gibt.

Unser Browser, also der Client, bereitet dann die Antwort auf, sodass wir die Trefferliste ansehen und durchscrollen können. Wollen wir die nächste Trefferseite aufrufen, startet unser Client die nächste Anfrage beim Server und holt Paket 2 ab.

Soweit verständlich, oder?

Für unser Spiel bedeutet das nun, dass wir uns um die Programmierung des Clients UND die des Servers kümmern müssen.

4.2.2 Das Framework

Hier müssen wir uns jetzt entscheiden, welches Framework wir einsetzen wollen.

Fragen wir und zuerst, was ist ein Framework und wozu brauchen wir das. Eine Definition könnte sein:

- Ein Framework ist ein vorgefertigtes Softwaregerüst, das Entwicklern Standardfunktionen und Strukturen bereitstellt, um Anwendungen schneller und effizienter zu entwickeln.

Okay, und welches Gerüst ist jetzt für unser Projekt geeignet? Wenn wir eine KI fragen, welches der beiden Frameworks Flask oder Django die größere Popularität hat, liegt Django mit 60-65% vorne.

Allerdings wird angemerkt, dass Flask für kleinere Projekte eher geeignet ist, da es leichtgewichtiger und flexibler ist und schneller zu erlernen sein soll.

Also genau unser Ding. Stürzen wir uns also auf **Flask**. Die Installation ist dann Teil von Kapitel 4.3.

4.2.3 Die Datenbank

Für die Auswahl der richtigen Datenbank fragen wir wieder unsere KI. Auf die Frage „welche Datenbank wird häufig in Flask-Anwendungen verbaut?“ bekommen wir den Namen „SQLite“ angezeigt. Gründe dafür sind:

- Es wird keine separate Serverkonfiguration benötigt
- Kann direkt in die Anwendung integriert werden
- Ist leichtgewichtig und
- gut für kleinere Projekte und Prototypen geeignet

Alternativ werden PostgreSQL und MySQL genannt, diese sollen besonders bei größeren Anwendungen zum Einsatz kommen.

Also gehen wir mit **SQLite**. Auch hier schauen wir uns die Verwendung im Kapitel 4.3 an.

4.2.4 Weitere Sprachen

Aus der Internetrecherche haben sich 4 Sprachen herauskristallisiert, die wir benutzen werden. Neben **Python** (dem Server) sind das **HTML** (der Client) mit **CSS** als Darstellungssprache, sowie **JavaScript** für die Kommunikation zwischen Client und Server. Nichts davon müssen wir extra installieren.

4.2.5 Noch etwas?

Ja. Nachdem nun unser Rahmen für die Entwicklung feststeht, fragen wir unsere KI nach einer Struktur für Flask mit einer SQLite-Datenbank, dem Frontend, CSS und JavaScript. Das Ergebnis sieht so aus:

```
mein_projekt/
│
├── instance/
│   └── database.db
│
├── static/
│   ├── css/
│   │   └── main.css
│   ├── js/
│   │   └── script.js
│   └── images/
│
├── templates/
│   ├── base.html
│   ├── index.html
│   └── andere_seiten.html
│
├── __init__.py
├── models.py
├── routes.py
├── config.py
├── requirements.txt
├── app.py
└── .gitignore
```

Sehr gut, das ist eine gute Basis. Die einzelnen Module klären wir gleich.

4.3 Das Coden

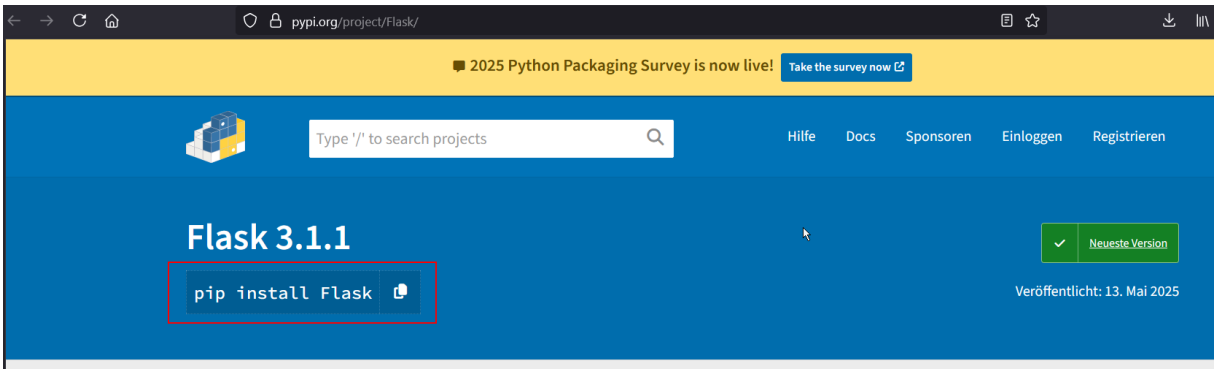
Puh, da haben wir uns ja echt was vorgenommen…

Damit wir auch verstehen, was wir tun, werden wir die einzelnen Komponenten getrennt voneinander betrachten und machen zuerst ein paar kleinere Übungen. Wenn wir uns dann sicher sind, bauen wir die einzelnen Komponenten zusammen.

Schauen wir uns also zuerst Flask an.

4.3.1 Arbeiten mit Flask

Wir haben gelernt, dass Flask ein Gerüst ist. Schauen wir uns auf der Startseite von Flask (https://pypi.org/project/Flask/) um, sehen wir direkt den Installationsbefehl mittels `pip`:

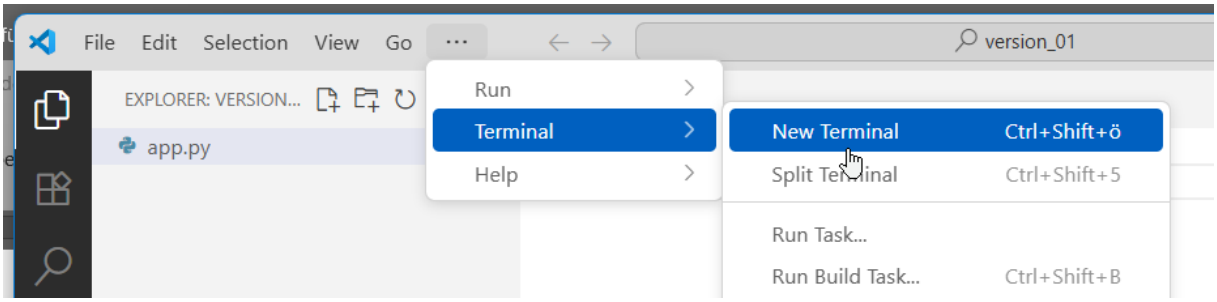


Bevor wir die Installation machen, sollten wir uns überlegen, ob wir für die Entwicklung unserer Anwendung eine eigene virtuelle Umgebung einrichten wollen oder nicht. Der Vorteil einer virtuellen ist ganz klar, dass wir alle Anforderungen nur in dieser Umgebung installieren können, was dann keine Auswirkung auf eine andere virtuelle Umgebung hat. Sollten wir also zwei unterschiedliche Anwendungen mit jeweils unterschiedlichen Softwareständen der gleichen Software nutzen, könnte das zu Problemen führen.

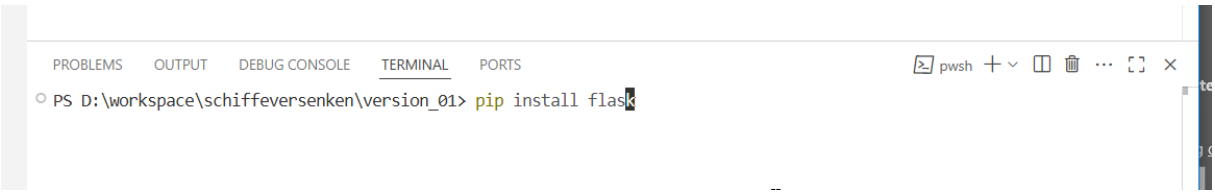
Mir ist das aber bisher noch nicht passiert, sodass ich die für mich einfachere Variante wähle und alle Software direkt installiere.

Solltet Ihr Euch für eine virtuelle Umgebung entscheiden, müsstet Ihr diese erst einrichten, bevor Ihr Flask installiert. Wie das geht, erfahrt Ihr am einfachsten über eine Internetrecherche…

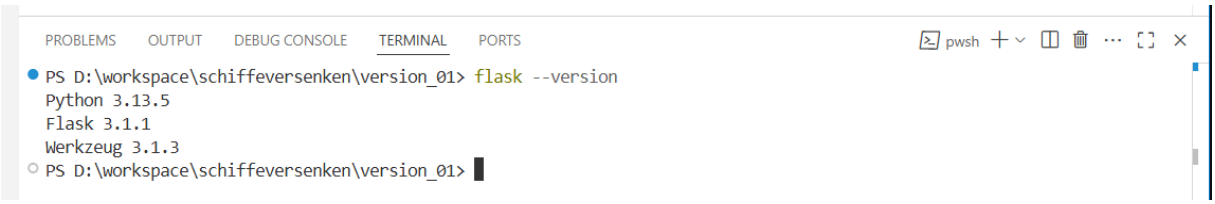
Für die Installation öffnen wir in VSCode über Terminal/New Terminal eine Eingabeaufforderung:



Hier geben wir jetzt den obenstehenden Befehl `pip install flask` ein und drücken Eingabe:



Die Installation geht fix, danach können wir mit Eingabe von `flask –version` den Erfolg kontrollieren:



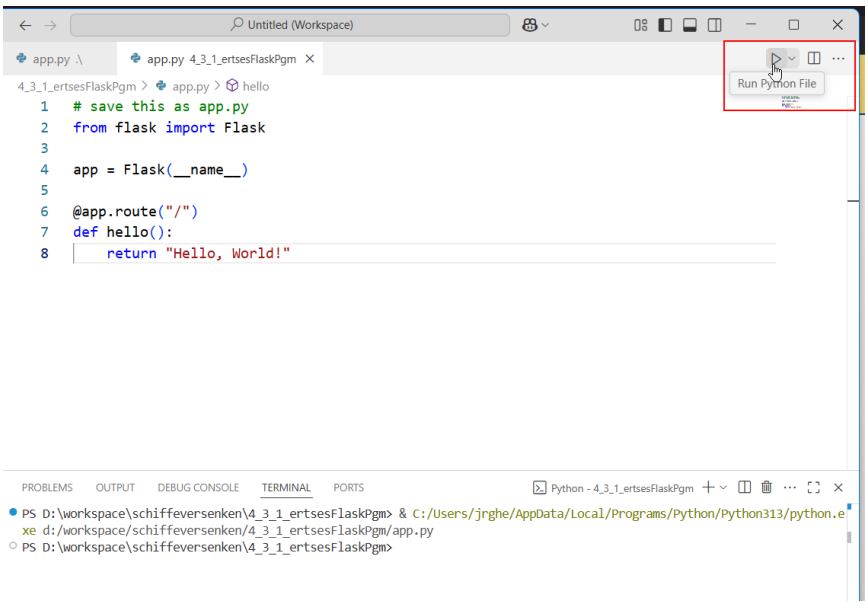
Bei Euch auch? Top!

Dann geht es jetzt los, lasst uns das erste Flask-Programm starten. Auf der Homepage von Flask ist gleich auf der ersten Seite etwas weiter unten ein Beispiel enthalten:

A Simple Example

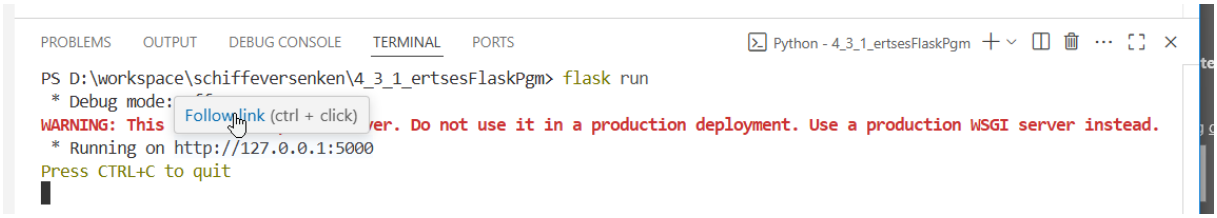
```
# save this as app.py
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello, World!"
```

Den Code schneiden wir uns aus und geben ihn in VSCode ein. Drücken wir dann den Run-Button oben rechts, passiert nicht viel:

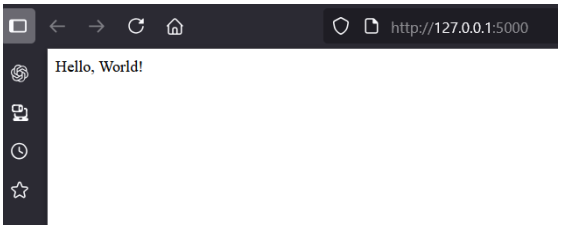


Enttäuschend, oder?

Das liegt daran, dass wir Flask mit dem Befehl `flask run` im Terminal starten müssen:



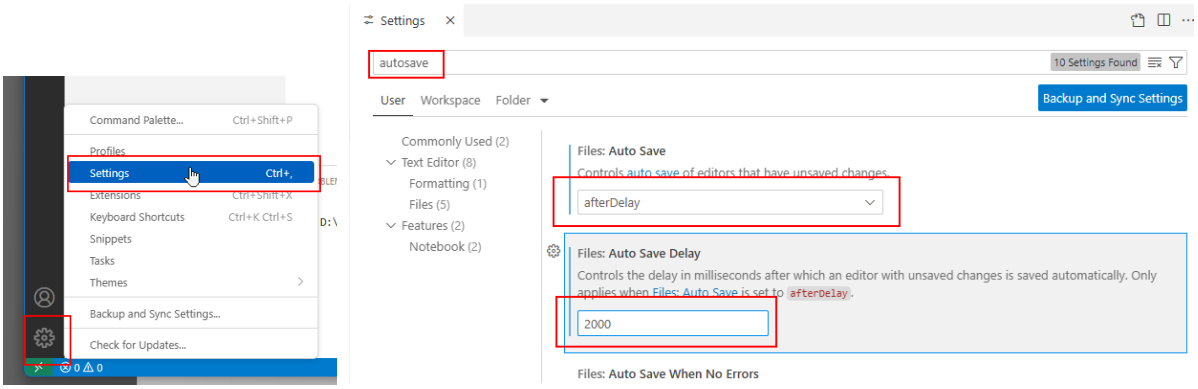
Wenn wir jetzt auf „Follow link“ klicken, öffnet sich der Standardbrowser und zeigt folgendes an:



Was ist passiert? Flask hat einen Server gestartet, der auf der IP-Adresse 127.0.0.1 und Port 5000 lauscht und eine URL bereitgestellt, die wir anklicken können.

Perfekt, das funktioniert also.

An dieser Stelle ein Tipp: VSCode bietet die Möglichkeit, Änderungen am Code eigenständig zu speichern. Die Einstellung erfolgt über Klick auf das Zahnrad unten links und dann die Settings aufrufen:



In das Suchfeld „`autosave`“ eingeben, dann erscheinen die beiden Reiter. Im ersten habe ich `afterDelay` ausgewählt, bedeutet, die automatische Speicherung erfolgt mit Zeitverzögerung. Diese kann man im unteren Reiter anpassen, ich habe sie auf 2000 Millisekunden, also 2 Sekunden gesetzt.

Für mich sehe ich den Vorteil darin, dass ich nicht explizit den Speichern-Button betätigen oder Strg+S drücken muss. Wenn die Änderung nicht zu einem Fehler in der Anwendung führt, bleibt die Serververbindung bestehen und ich brauche nur den Browser zu aktualisieren um die Änderung zu sehen.

Kachelt die Verbindung ab, müsst Ihr den Server erneut starten. Stoppen könnt Ihr den Server entweder durch Klick auf das Löschen-Ikon des Terminalfensters, oder durch Eingabe von „STRG+C“.

Wenden wir uns jetzt der vorgeschlagenen Programmstruktur zu.

Wir haben uns ja vorgenommen, dass wir langsam anfangen. Daher beschränke ich mich jetzt mal auf die rudimentären Module und lasse Datenbank und andere Module erst einmal außen vor.

Das, was wir jetzt einrichten werden sieht so aus:

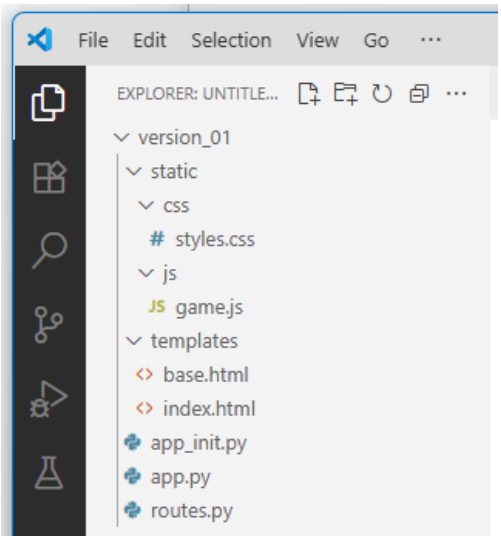
```
schiffeversenken/version_01/
│
├── static/
│   ├── css/
│   │   └── styles.css
│   └── js/
│       └── game.js
│
├── templates/
│       ├── base.html
│       └── index.html
│
├── app_init.py
├── app.py
└── routes.py
```

Wir haben unsere Projektseite, darin die 2 Ordner `static` und `templates`, sowie das Startmodul `app.py` und die Konfigurationsdatei `app_init.py` und das Modul `routes.py`.

Der Ordner `static` hat 2 Ordner, den Ordner `css` mit unserer `styles.css` für das Aussehen der HTML-Seiten und den Ordner `js` der unsere JavaScript-Datei enthält. Im Ordner `templates` liegen die HTML-Dokumente.

Lasst uns gemeinsam die Struktur anlegen, damit wir auf dem gleichen Stand sind. Ich lege in VSCode erst alle Ordner an (Rechtsklick auf den Ordner im Workspace, dann New Folder). Die Dateien lege ich auch gleich alle an (Rechtsklick auf den jeweiligen Unterordner, dann New File). Bei der Anlage der Files muss die Endung explizit mit eingegeben werden. Also „`game`“ reicht nicht, man muss „`game.js`“ eingeben.

Bei mir sieht das dann so aus:



Bei Euch auch? Schön so! Ich würde vorschlagen, ich gebe Euch gleich mal Kopiervorlagen für einige der Dokumente und dann schauen wir uns das gemeinsam an.

In die `app.py` kopiert mal bitte folgendes:

```
from app_init import create_app

app = create_app()

if __name__ == '__main__':
    app.run(debug=True)
```

In die `app_init.py` kommt das hier rein:

```
from flask import Flask

def create_app():
    app = Flask(__name__)

    # Blueprints registrieren
    from routes import main
    app.register_blueprint(main)

    return app
```

Weiter geht es mit der `routes.py`:

```
from flask import Blueprint, render_template

# Erstelle den Blueprint für die Routen
main = Blueprint('main', __name__)

@main.route('/')
def index():
    return render_template('index.html')
```

Die Datei `base.html` wird hiermit bestückt:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Schiffeversenken</title>
    <link
      rel="stylesheet"
      href="{{url_for('static', filename='css/styles.css')}}"
    />
      <link
href="https://fonts.googleapis.com/css2?family=Montserrat:wght@400&display=swap"
rel="stylesheet">

  </head>
  <body>
    <div class="content">{% block content %} {% endblock %}</div>
  </body>
</html>
```

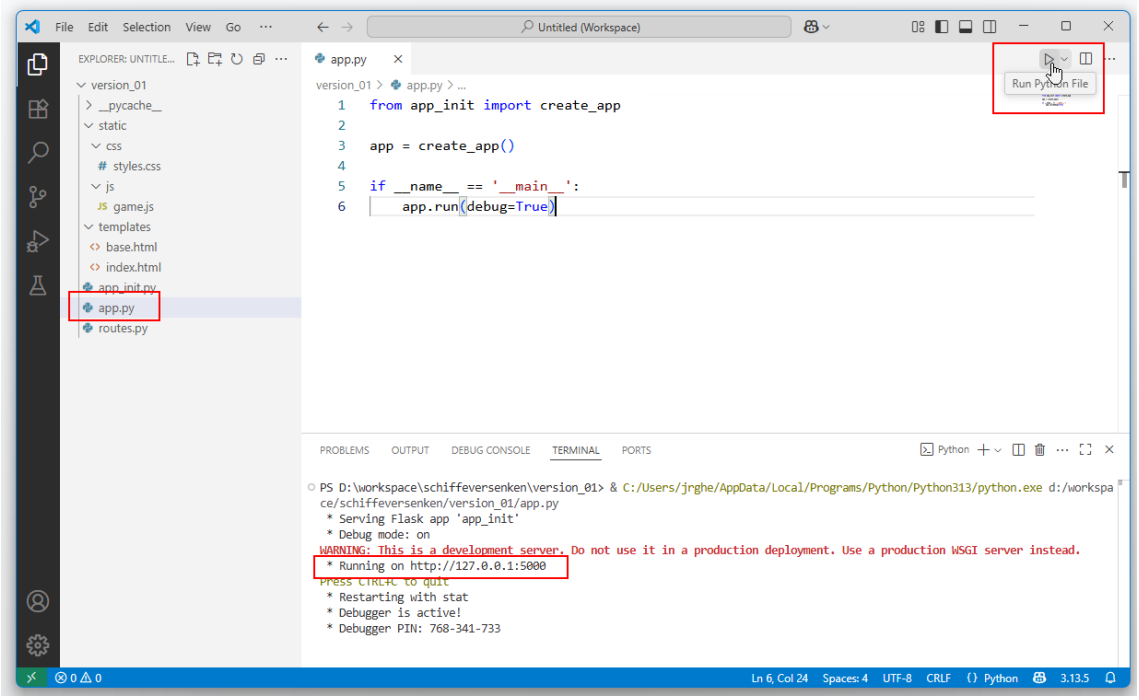
In `index.html` kommt:

```
{% extends 'base.html' %} {% block content %}
<h3>Kann losgehen?</h3>
<div class="button-text-container">
    <button type="submit" class="button-index">Go</button>
</div>
{% endblock %}
```

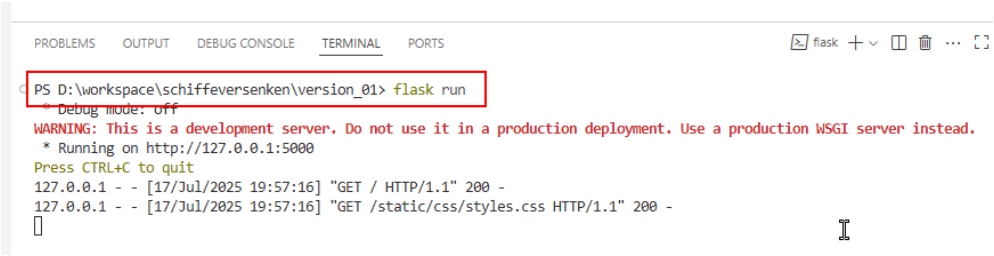
x:

```
/* Grundlayout */
body {
  font-family: 'Montserrat', Verdana, Geneva, Tahoma, sans-serif; /* Schriftart */
  background-color: #f8f9fa;
  margin: 0;
  padding: 20px;
  color: #333;
}

/* Button */
.button-index {
  padding: 10px 20px;
  margin: 5px;
  background-color: #007bff;
  border: none;
  color: white;
  font-weight: bold;
  border-radius: 5px;
  cursor: pointer;
}
```

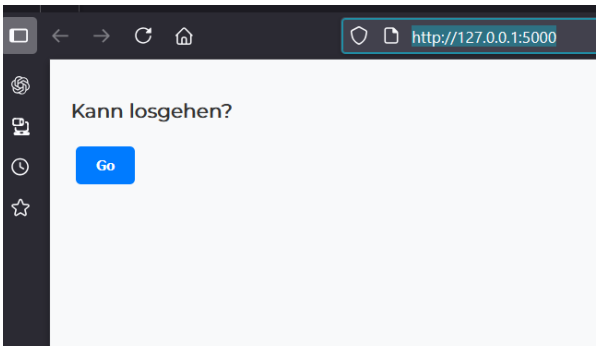
Die Datei `games.js` bleibt erst einmal leer. Jetzt müsste bei Euch auch wieder der Run-Button oben rechts funktionieren:



Falls nicht, Starten der Anwendung über ein neues Terminalfenster, dann `flask run` eingeben, das sollte es auch tun:



Nun wieder dem Link folgen oder im Browser http://127.0.0.1:5000/ eingeben, dann landet Ihr auf der Startseite der Anwendung:



Sieht bei Euch auch so aus? Falls nicht, müsstet Ihr prüfen, ob Ihr alle Kopien am richtigen Ort abgelegt habt.

Dann lasst uns die Verarbeitung mal durchgehen, das ist wichtig für das Verständnis des Zusammenspiels der einzelnen Module.

```
1   from app_init import create_app
2
3   app = create_app()
4
5   if __name__ == '__main__':
6       app.run(debug=True)
```

Fangen wir im Start-Modul `app.py` an. Zunächst importieren wir aus `app_init.py` die Funktion `create_app()` und erzeugen die Instanz `app`.

Zeile 5 ist der Programmstart mit dem Befehl `app.run` im Debug-Modus

Weiter geht es mit der `app_init.py`.

```
1   from flask import Flask
2
3   def create_app():
4       app = Flask(__name__)
5
6       # Blueprints registrieren
7       from routes import main
8       app.register_blueprint(main)
9
10      return app
```

Auch hier zunächst die Importanweisungen, hiermit binden wir `Flask` ein.

In der Funktion `create_app()` erzeugen wir dann die Instanz einer Flask-App, stellen die Verbindung zu den HTML-Seiten her indem wir das `routes.py`-Modul ansprechen und von dort die eingetragenen main-Routen importieren.

Die Funktion gibt dann die app-Instanz zurück.

```python
from flask import Blueprint, render_template

# Erstelle den Blueprint für die Routen
main = Blueprint('main', __name__)

@main.route('/')
def index():
    return render_template('index.html')
```

Damit kommen wir dann schon zu den Routen, die alle im Modul `routes.py` abgelegt werden. Wir werden das gleich erweitern, dann wird das klarer.

Von Flask besorgen wir uns im Import die Funktionen `Blueprint` (also „Bauplan“) und `render_template`, die für die Aufbereitung der Inhalte auf der HTML-Seite sorgt.

Damit sind wir dann auch schon bei den beiden Seiten `base.html` und `index.html`. Fangen wir mit `base.html` an:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Schiffeversenken</title>
    <link
      rel="stylesheet"
      href="{{url_for('static', filename='css/styles.css')}}"
    />
      <link href=
      "https://fonts.googleapis.com/css2?family=Montserrat:wght@400&display=swap"
      rel="stylesheet">

  </head>
  <body>
    <div class="content">{% block content %} {% endblock %}</div>
  </body>
</html>

```

Wer HTML und **Jinja2** kennt, wird hier nicht überrascht sein. Wer HTML kennt, aber Jinja2 nicht, wird sich vielleicht über Zeile 16 wunden, ansonsten ist nichts Auffälliges. Wer aber HTML noch nicht kennt, für den ist alles neu.

Für die Anfänger unter Euch – sehr vereinfacht, die Syntax von HTML besteht in der Regel aus dem öffnenden `<>` und dem schließenden `</>`, eckigen Klammerpaar, den „`tags`“. Alles was dazwischen steht, wird als dem `tag`-zugehörig interpretiert. Schaut Euch die Zeilen 2 und 18 an (`<html lang="en">` und `</html>`) oder 3 und 14 (`<head>` und `</head>`), dann wird das deutlich. Nicht immer sind schließende Klammerpaare notwendig, manchmal steht das Endezeichen „/“ mit im öffnenden Klammerpaar (wie in Zeile 6 bis 9, der Link auf unser CSS-Dokument) oder es fällt gänzlich weg (wie im nächsten Link).

Die beiden großen Blöcke sind `<head>` und `<body>`. In `<head>` sind die Steuerinformationen enthalten, also zum Beispiel der Titel, der oben im Reiterkopf des Browsers angezeigt wird, aber auch die Links zu der CSS-Datei oder auch das Einbinden einer weiteren Schriftart, in unserem Fall „Montserrat“ von Google. Der `<body>`-Tag enthält dann alle Informationen, die im Inneren des Browserfensters angezeigt werden.

Damit kommen wir zur Besonderheit in Zeile 16 – Jijna2. Jijna2 ist eine für Python entwickelte „Platzhaltersprache“, die ermöglicht, dynamisch Code in HTML-Seiten einzufügen.

Die beiden geschweiften Klammerpaare `{% block content %}` und `{% endblock %}` sind solche Jinja2-Befehle und zeigen der Seite an, dass der konkrete Inhalt durch einen anderen Prozess geliefert wird. In unserem Fall ist das die `index.html`-Seite:

```
1  {% extends 'base.html' %} {% block content %}
2  <h3>Kann losgehen?</h3>
3  <div class="button-text-container">
4      <button type="submit" class="button-index">Go</button>
5  </div>
6  {% endblock %}
7
```

Die beiden Zeilen 1 und 6 sind die Jinja2-Befehle, die die Inhalte aus der Datei `base.html` zufügen. Die `index.html` wird dann also inklusive der `head`-Daten angezeigt.

Wir hätten auch alles in einer HTML-Datei zusammenfassen können, dann sähe das so aus:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Schiffeversenken</title>
6      <link
7        rel="stylesheet"
8        href="{{url_for('static', filename='css/styles.css')}}"
9      />
10       <link href=
11       "https://fonts.googleapis.com/css2?family=Montserrat:wght@400&display=swap"
12       rel="stylesheet">
13
14   </head>
15   <body>
16     <div class="content">
17         <h3>Kann losgehen?</h3>
18         <div class="button-text-container">
19             <button type="submit" class="button-index">Go</button>
20         </div></div>
21   </body>
22 </html>
23
```

Probiert das gerne aus! Der Vorteil von Jinja2 tritt erst zutage, wenn man mehr als eine HTML-Seite hat. Da das bei uns der Fall sein wird, bleiben wir bei Jinja2.

Schauen wir uns als letztes noch kurz die Datei `styles.css` an:

```
1  /* Grundlayout */
2  body {
3    font-family: 'Montserrat', Verdana, Geneva, Tahoma, sans-serif;
4    background-color: #f8f9fa;
5    margin: 0;
6    padding: 20px;
7    color: #333;
8  }
9
10 /* Button */
11 .button-index {
12   padding: 10px 20px;
13   margin: 5px;
14   background-color: #007bff;
15   border: none;
16   color: white;
17   font-weight: bold;
18   border-radius: 5px;
19   cursor: pointer;
20 }
21
```

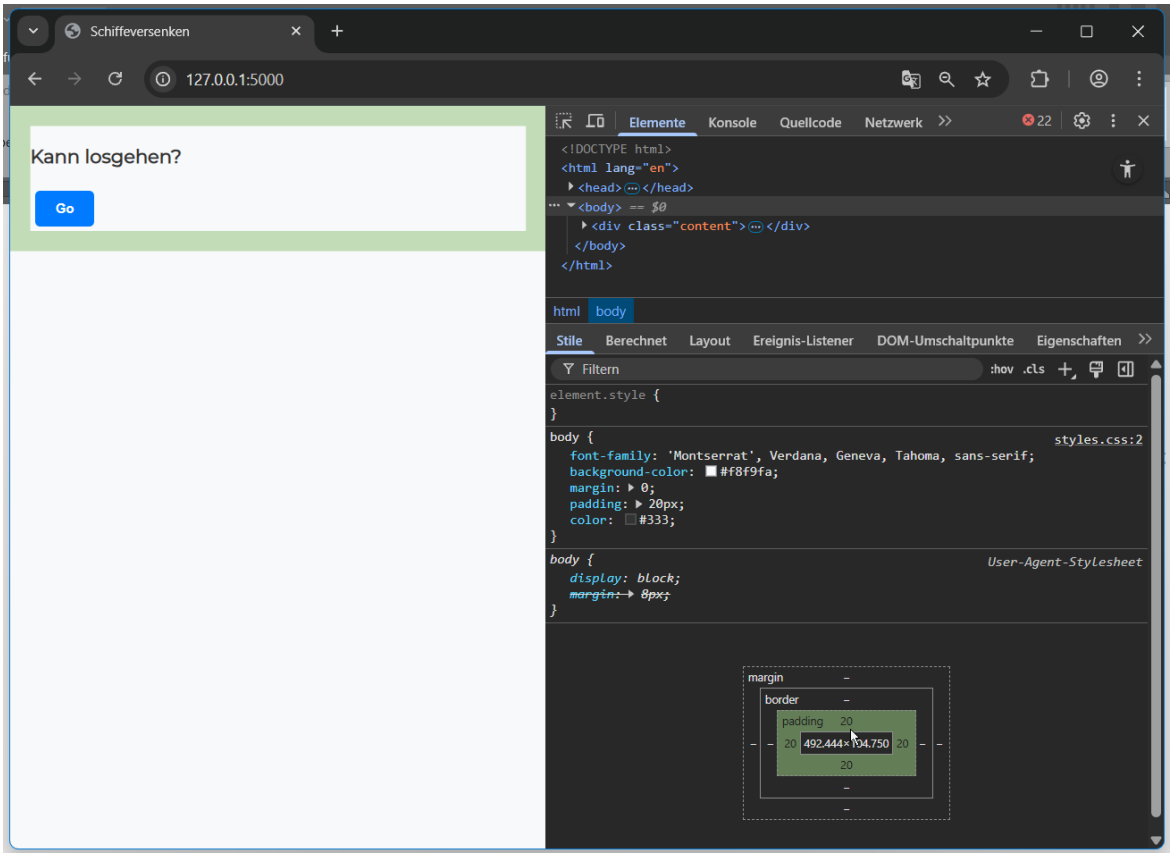
Hier definieren wir das Aussehen und das Interagieren unserer einzelnen HTML-Komponenten.

Mit „`body`“ wird alles angesprochen, was in der `base.html` zwischen den Zeilen 15 und 17 passiert also das, was mit Jinja2 aus der `index.html` gezogen wird.

In Zeile 4 in der `index.html` ist ein Button definiert, dem wurde die Klasse „`button-index`“ zugewiesen. Diese Klasse wird in der `styles.css` in Zeile 11 definiert.

Nicht definierte Klassen wie in Zeile 3 der `index.html` (`class="button-text-container"`) werden vom Browser ignoriert.

Ihr könnt Euch das auch genauer anschauen, wenn Ihr den Server gestartet habt und die Funktionstaste `F12` drückt, seht Ihr die Informationen der Seite:



Da sieht man auch ganz gut, was die unterschiedlichen Einträge bewirken.

Damit haben wir die wesentlichen Konzepte **Flask** und **Jijnja2** besprochen, vertiefende Informationen gibt es haufenweise im Netz.

Im nächsten Kapitel nehmen wir die Datenbank dazu.

4.3.2 SQLite

4.3.2.1 Vorüberlegungen

Was wollen wir mit einer Datenbank anfangen und warum machen wir das überhaupt? Und wie funktioniert das technisch?

Diesen Fragen widmen wir uns jetzt, bevor wir ans Coden gehen.

Der Begriff „`Datenbank`“ wird oft synonym mit „`Datenbanktabelle`“ oder kurz nur „`Tabelle`“ genutzt. Die Daten selbst liegen in der Datenbanktabelle. Die Organisation der Datenbanktabelle obliegt dabei dem Datenbanksystem, also in unserem Fall `SQLite`. In einer **Datenbank** können viele verschiedene **Datenbanktabelle** gespeichert sein, diese sollten allerdings immer einen fachlichen Bezug untereinander haben.

Ich habe mal ein Beispiel einer Autovermietung gemacht, da hatten wir eine Datenbanktabelle mit den Informationen zum **Kunden** eine weitere mit den Informationen zu unserem **Fuhrpark** und eine dritte Datenbanktabelle zur **Vermietung** selbst. Also 3 Datenbanktabellen in einer Datenbank.

Machen wir es für unser Projekt konkret, die Datenbank wird bei uns `battle.db` heißen, die Datenbanktabelle in der die Informationen abgelegt sind, heißt `game`.

Wie kommen wir jetzt aber zum Aufbau und den konkreten Feldern in der Datenbanktabelle?

Uns geht es ja um die Ablage von strukturierten Daten in einem Container, den wir jederzeit befüllen, leeren und befragen können wollen.

Sammeln wir also zunächst die Informationen, die wir in der Datenbanktabelle ablegen wollen.

Wir haben

- 2 Spieler, die sich
- in einem Spiel treffen,
- jeder Spieler hat sein eigenes Spielfeld und
- sie spielen abwechselnd gegeneinander

Vermutlich fallen uns zu einem späteren Zeitpunkt noch weitere Dinge auf oder ein. Gehen wir es durch.

Spieler werden durch ihre Namen repräsentiert. Wir könnten uns eine Anmeldefunktion mit einer eigenen Datenbanktabelle für Nutzer-Passwort anlegen, das ist aber etwas für eine zukünftige Erweiterung. Für jetzt geben wir dem Spieler die Möglichkeit, einen Namen einzugeben, beim nächsten Spiel kann er sich anders nennen.

Wie können wir sicherstellen, **dass sich 2 Spieler in einem Spiel treffen** können? Wir brauchen eine eindeutige Kennung, die das Spiel repräsentiert und die die beiden Spieler untereinander austauschen.

Ich habe dazu eine tolle Idee von Tim in einem Video seiner Reihe `techwithtim` gesehen, die wir adaptieren werden. Jedes Spiel bekommt einen 6-stelligen Code bestehend aus 6 zufällig zusammengewürfelten Großbuchstaben. Es wird geprüft, ob der Code schon in der Datenbank existiert, falls ja, wird ein neuer Code generiert. Damit erreichen wir Eindeutigkeit und haben nach meinen Berechnungen rund 308 Millionen verschiedene Möglichkeiten. Das sollte fürs erste reichen…

Wie können wir die **Spielfelder** abbilden? Jedes Spielfeld ist 10 Felder breit und 10 Felder hoch, besteht also aus 100 Feldern. Wir nehmen uns also einen 100 Byte langen String, in dem jede Stelle ein Feld repräsentiert.

Es gibt 4 Zustände, die ein einzelnes Feld haben kann:

- es ist ein leeres Feld,
- es befindet sich ein Schiff darauf,
- oder es hat einen Treffer bekommen und war vorher ein leeres Feld
- oder Teil eines Schiffes.

Ein leeres Feld belegen wir mit einer „0“, ein Schiff können wir mit einer „1“ belegen. Die Schüsse markieren wir dann mit „T“ für Treffer oder „W“ für Wasser.

Kommen wir zur letzten Frage, wie können wir mit den uns vorliegenden Informationen sagen, welcher **Spieler am Zug** ist? Oder wer am Ende gewonnen hat?

Noch nicht, daher brauchen wir noch mindestens ein Feld, das Statusinformationen aufnimmt. Also „Spieler 1 hat das Spiel eröffnet“, „Spieler 2 ist beigetreten“ oder „Spieler 1 hat gewonnen“.

Alternativ können wir jedem Spieler auch einen eigenen Status geben, das könnte die Steuerung einfacher machen, da an jedem Spieler individuelle Aktionen gespeichert werden können.

Mit einem oder 2 Statusfeldern sind wir dann in der Lage, die beiden Zustände „Spieler 1 am Zug“ oder „Spieler 2 am Zug“ abbilden zu können. Ich entscheide mich mal für 2 separate Statusfelder.

Prima, damit sollten wir alle Informationen zusammen haben. Wie kommen wir aber jetzt zu einer konkreten Datenbanktabelle?

Tragen wir die Einzelteile zuerst in einer „normalen“ Tabelle zusammen:

Subjekt	Erklärung
Kennung des Spiels	Ein Spiel besteht aus genau 2 Spielern, und diese müssen sich über eine eindeutige Kennung des Spiels zusammenfinden
Name Spieler 1	Name des ersten Spielers
Spielfeld Spieler 1	Informationen zum Spielfeld, also Position der Schiffe, Treffer- oder Wasserschüsse, des Gegners
Status Spieler 1	z.B. „Spiel eröffnet“, „wartet auf Zug“, „hat gewonnen“
Name Spieler 2	Name des zweiten Spielers
Spielfeld Spieler 2	Wie oben, allerdings die Abbildung der gegnerischen Schiffe
Status Spieler 2	z.B. „Spiel beigetreten“, „am Zug“, „hat verloren“

Wenn wir diese Tabelle jetzt „kippen“ – also Spalten werden zu Zeilen – dann haben wir schon die Struktur für eine Datenbanktabelle. Geben wir noch eine eindeutige ID dazu und füllen das gleich noch mit ein paar hypothetischen Inhalten, sehen wir auch, ob das Konzept passt:

ID	Kennung des Spiels	Name Spieler 1	Spielfeld Spieler 1	Status Spieler 1	Name Spieler 2	Spielfeld Spieler 2	Status Spieler 2
1	ABCDEF	Adam	1100TW…	am Zug	Adele	1111WW…	wartet
2	GHIJKL	Berta	1111100…	wartet	Ben	WW0000…	am Zug
3	MNOPQ	Casper	Spiel gestartet				

Kurze Prüfung

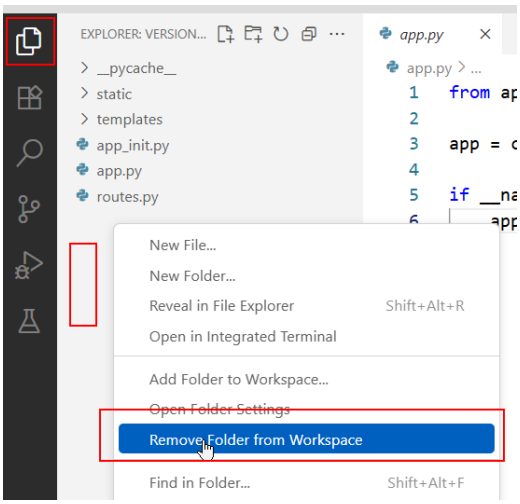
- Spiel ist eindeutig identifizierbar
- Identifikation Spieler 1 und Spieler 2 ist möglich
- Identifikation Spielfeld von Spieler 1 und von Spieler 2 ist möglich
- Identifikation aktueller Spieler ist möglich

Okay, sehr cool, damit arbeiten wir weiter. Falls uns in der Entwicklung auffällt, dass etwas fehlt, erweitern wir die Tabelle.

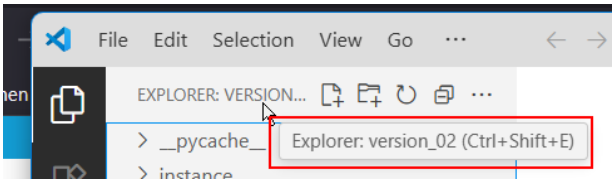
4.3.2.2 Implementierung

Bevor wir an die Implementierung gehen, sollten wir uns eine neue Version erstellen, damit wir zur Not auf die Vor-Version zurückgreifen können.

Ich kopiere dazu den gesamten Ordner `version_01` und füge ihn als `version_02` wieder ein. Da wir die `version_01` nicht mehr in unserem VSCode brauchen, entferne ich sie über Rechtsklick unterhalb der Elemente in der Explorer-Ansicht und dann „Remove…“:



Die neue Version im Ordner `version_02` dann wie oben beschrieben in VSCode aufnehmen:



Was brauchen wir jetzt Code-seitig? Wir brauchen folgende Erweiterungen (neu = rot umrandet):

```
schiffeversenken/version_02/
│
├── instance/
│   └── battle.db
│
├── static/
│   ├── css/
│   │   └── styles.css
│   └── js/
│       └── game.js
│
├── templates/
│       ├── base.html
│       └── index.html
│
├── app_init.py
├── app.py
├── models.py
├── routes.py
└── script.py
```

Wir erstellen den Ordner `instance`, die Datenbank wird dann durch die Anwendung selbst erzeugt.

Zusätzlich brauchen wir noch 2 Module, `models.py` und `script.py`.

Das Modul `models.py` enthält die Klassendefinition, das `script.py` ist eine Hilfe zur Erzeugung und Pflege der Datenbank.

Erweitert wird das Modul `app_init.py`.

Nachdem wir die leeren Hüllen angelegt haben, geht es weiter mit `app_init.py`.

Der neue Code in der app_init.py ist folgender:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

# Erstelle die db Instanz
db = SQLAlchemy()

def create_app():
    app = Flask(__name__)

    # Konfiguration laden
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///battle.db'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    app.config['SECRET_KEY'] = 'IrgendeinWortOderEineZahlWie42'

    # Datenbank initialisieren
    db.init_app(app)

    # Blueprints registrieren
    from routes import main
    app.register_blueprint(main)

    return app
```

Von Flask holen wir uns `flask_sqlalchemy`, laden die Konfiguration mit Ablageort, das interne Tracking, und einen Schlüssel-Wert zum Zugriff auf die Tabelle. Hier könnt Ihr irgendwas eintragen.

Neu ist jetzt `models.py`. Hier tragen wir die Datenbanktabellenfelder als Klasse ein:

```
from app_init import db

class Game(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    game_code = db.Column(db.String(6), nullable=False)
    p_1_name = db.Column(db.String(100), nullable=True)
    p_1_board  = db.Column(db.String(100), nullable=True)
    p_1_status = db.Column(db.String(100), nullable=True)
    p_2_name = db.Column(db.String(100), nullable=True)
    p_2_board = db.Column(db.String(100), nullable=True)
    p_2_status = db.Column(db.String(100), nullable=True)

    def __repr__(self):
        return f"{self.game_code}"
```

Auch hier wieder zuerst der Import der `db` aus `app_init` und dann die Klasse `Game` mit den Feldern aus unserer Tabelle oben. Die Längenangaben in den Strings sind keine wirklich festen Werte, `SQLAlchemy` konvertiert Strings für `SQLite`. Wichtig für jede Datenbanktabelle sind die Angaben des primären Schlüssels, dieser darf niemals doppelt in der Datenbanktabelle vorkommen, sonst ist sie „korrupt“ und damit nicht mehr zu benutzen.

Ich habe mich hier für eine `id` entschieden, diese wird automatisch vom System vergeben. In anderen Datenbanksystemen muss man dafür explizite Angaben wie „`AUTOINCREMENT = TRUE`“ oder ähnliches machen.

Der Zusatz `nullable=False` bedeutet, dass das Feld bei Speicherung nicht leer sein darf. Das gilt für unser Spiel für die 3 Felder `game_code`, `p_1_name` und `p_1_status`. Die restlichen Felder können bei der ersten Speicherung leer sein.

Damit kommen wir zum Modul `script.py`. Nachfolgenden Code könnt Ihr einfach übernehmen:

```
from app_init import create_app, db
from models import Game
from sqlalchemy import create_engine, inspect

# Initialwerte
game_code_init = 'ABCDEF'
p_1_name_init = 'Adam'
p_1_board_init  = '1000T0…'
p_1_status_init = 'am Zug'
p_2_name_init = 'Adele'
p_2_board_init = '1110T0…'
p_2_status_init = 'wartet'

app = create_app()
engine = create_engine('sqlite:///instance/battle.db', echo=True)

def drop_table():
    inspector = inspect(engine)
    if 'game' in inspector.get_table_names():
        Game.__table__.drop(engine)
        print("Tabelle 'Game' wurde gelöscht.")
    else:
        print("Tabelle 'Game' existiert nicht, Step wird übersprungen.")

def create_table():
    with app.app_context():
        try:
            db.create_all()  # Erstellt alle Tabellen
            print("Tabellen wurden erstellt.")
        except Exception as e:
            print(f"Fehler beim Erstellen der Tabellen: {e}")
            return  # Funktion abbrechen, wenn Tabellen nicht erstellt werden konnten

        try:
            new_game = Game(game_code   = game_code_init,
                            p_1_name    = p_1_name_init,
                            p_1_board   = p_1_board_init,
                            p_1_status  = p_1_status_init,
                            p_2_name    = p_2_name_init,
                            p_2_board   = p_2_board_init,
                            p_2_status  = p_2_status_init
                            )
            db.session.add(new_game)
            db.session.commit()
            print("Initiales Spiel erfolgreich angelegt.")
        except Exception as e:
            db.session.rollback()  # wichtig: Rollback bei Fehlern
            print(f"Fehler beim Einfügen des initialen Spiels: {e}")

if __name__ == '__main__':
    print('Start')
    drop_table()
    create_table()
    print('Ende')
```

Zuerst wieder die Importe, dann die Belegung der Initialwerte mit den Daten der ersten Zeile unserer Tabelle oben. Programmstart ist ganz unten (`if __name__ == '__main__':`), es werden nacheinander die beiden Funktionen `drop_table()` und `create_table()` aufgerufen.

Mit `DROP_TABLE` wird in der Sprache SQL eine Tabelle physisch gelöscht (also nicht „geleert“) der Erstellungsbefehl einer Tabelle ist demnach `CREATE_TABLE`.

Die entscheidenden beiden Zeilen sind:

```
db.session.add(new_game)
db.session.commit()
```

Wir müssen keinen eigenen SQL-Befehl zum Speichern der Daten absetzen, es reicht, wenn wir eine Instanz der Klasse `Game` erzeugen, diese der `session` hinzufügen und dann mit dem der Funktion `commit()` bestätigen. Ohne den `commit`-Befehl werden Einträge in der Datenbank zwar gespeichert, sind für andere Prozesse aber noch nicht sichtbar. Es gibt zwar Techniken, wie man auch an diese Sätze kommt, das führt hier aber zu weit.

Drei Zeilen darunter gibt den Aufruf der Funktion `rollback()`:

```
db.session.rollback()  # wichtig: Rollback bei Fehlern
```

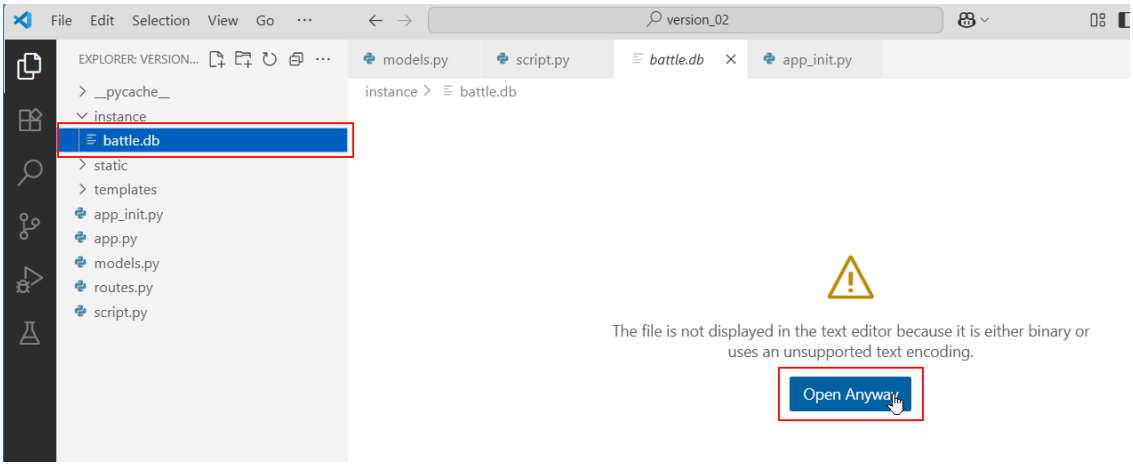
Dieser macht die Änderungen rückgängig, die sich in dem Zwischenzustand vor Aufruf des `commit()` befinden.

Okay, cool, showtime!

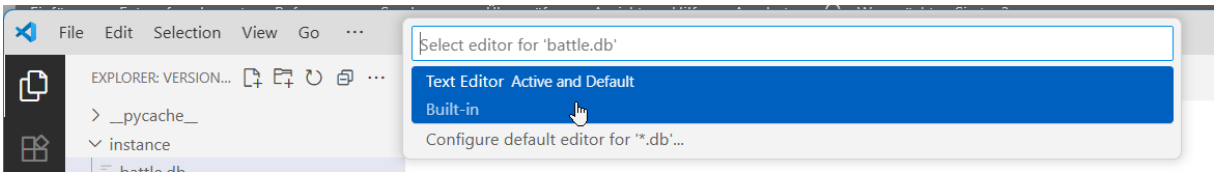
Da wir mit „`if __name__ == '__main__':`“ den autonomen Programmstart des Skripts festgelegt haben, können wir das Skript einfach mit dem Run-Button oben rechts starten. Bei mir kommt dann ein Terminal-Fenster mit der Info:

Das sieht doch super aus, dann lasst uns mal in die Datenbank schauen.

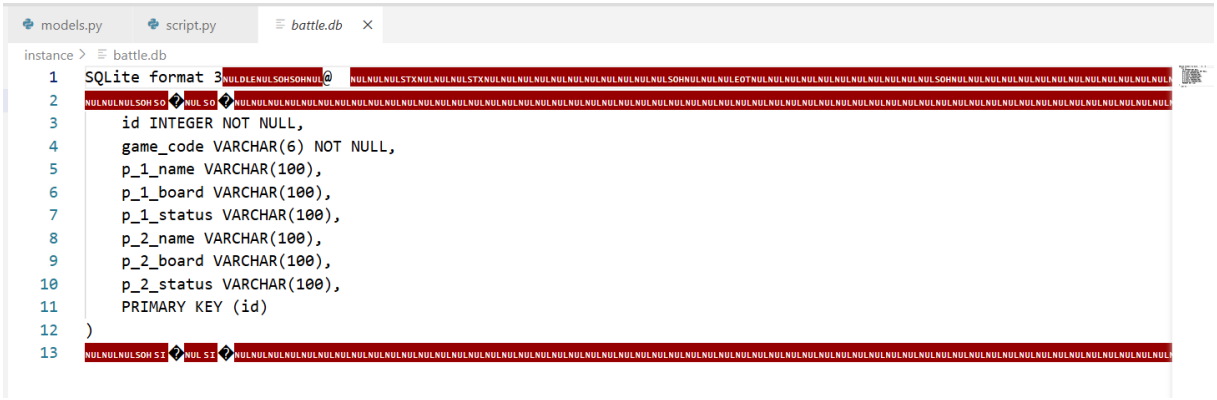
In VSCode ist das schwierig. Wenn wir auf die `battle.db` klicken, sagt uns VSCode, dass es den Inhalt nicht anzeigen kann:



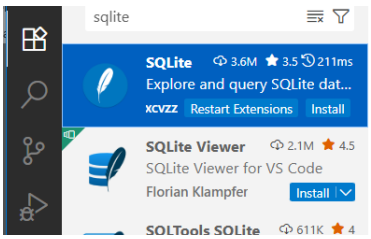
Klicken wir auf „Open Anyway“ und dann auf den ersten Eintrag oben,

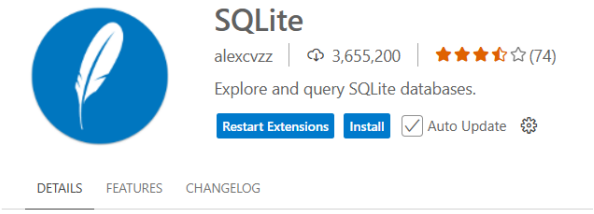


erhalten wir folgende Infos:



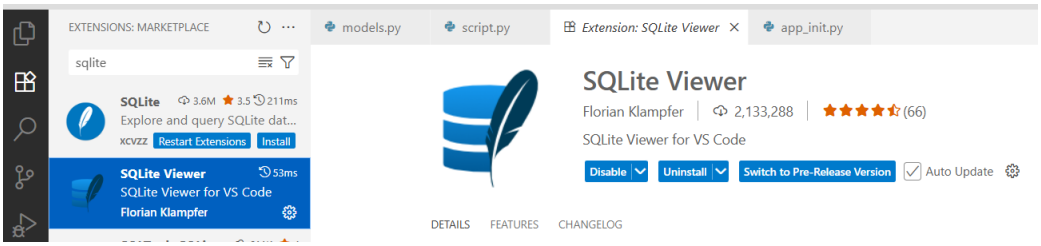
Okay, die Struktur scheint zu passen, aber wo ist der Datensatz? Es gibt eine Erweiterung in VSCode, für SQLite, die habe ich aber nicht zum Fliegen gebracht:





Vielleicht habt Ihr ja mehr Glück und könnt das Plugin erfolgreich einbinden, dann schreibt mir gerne Eure Lösung. Irgendwie muss das funktionieren, sonst wäre es nicht 3,6 Millionen Mal heruntergeladen worden und eine Bewertung von 3,5 Sternen…

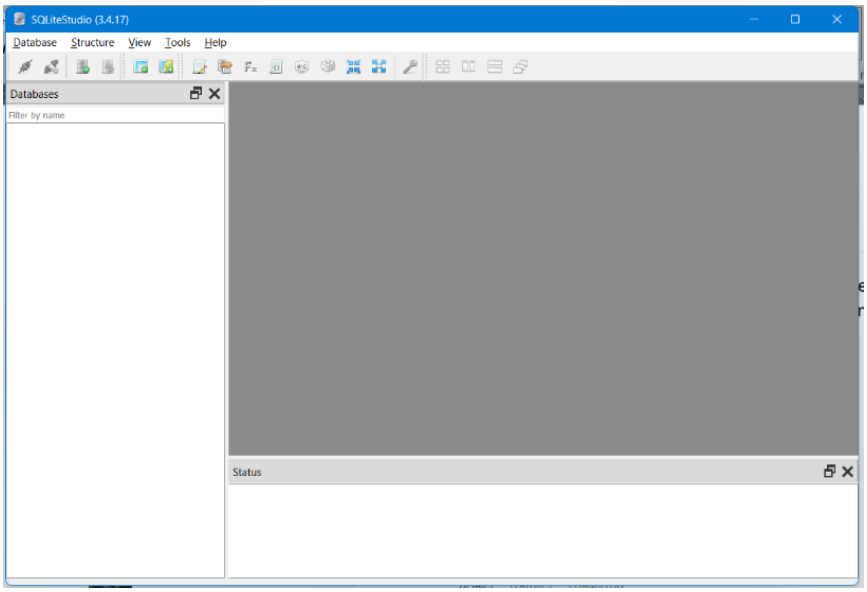
Es gibt noch das Plugin „SQLite Viewer“, das habe ich installiert:



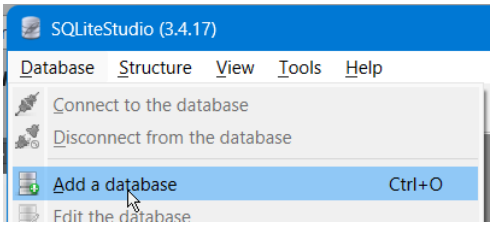
Damit kann man allerdings „nur“ Anschauen, um Änderungen an der Datenbank oder den Datenbanktabellen machen zu können, braucht man die Pro-Version. Diese kostet einmalig ab $79, das brauche ich nicht.

Als kostenlose Alternative habe ich `SQLiteStudio` installiert, damit bin ich bis jetzt noch an keine Grenze gestoßen.

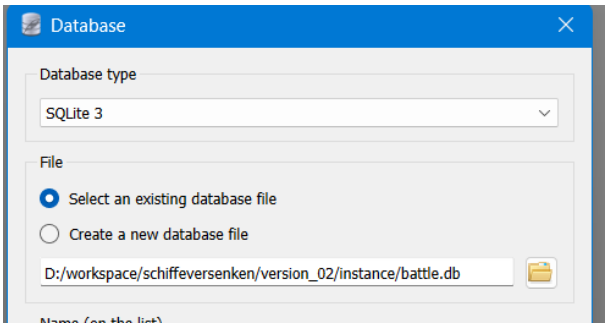
Die Installation von oben hat ja funktioniert, der erste Aufruf sollte bei Euch dann etwa so aussehen:



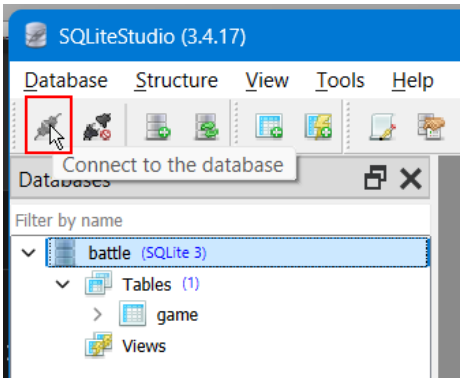
Mit “Database/Add a database”



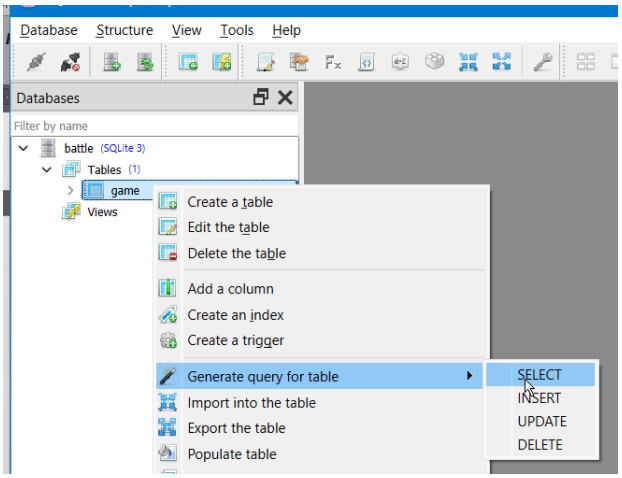
und Navigation zu unserer `battle.db` in der `version_02`, …

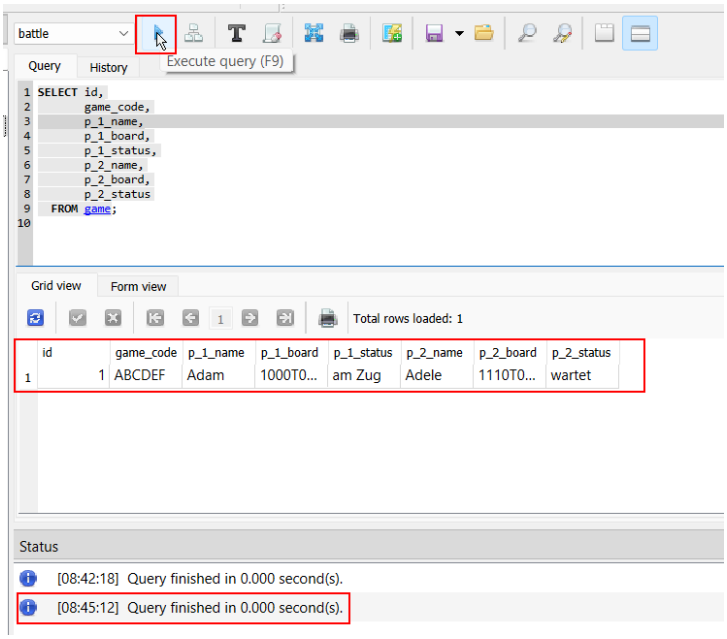


… dann oben links auf „Connect to database”…



… und Rechtsklick auf `game` ein Select-Fenster aufmachen:





Im Fenster rechts dann erscheint die Abfrage mit allen Feldern.

Oben auf den blauen Pfeil klicken und die Abfrage wird ausgeführt.

In der Mitte sieht man dann das Ergebnis der Abfrage, unsere Daten sind angekommen, also alles okay!

0,000 Sekunden – das ist mal schnell…

Wunderbar, damit haben wir die Datenbank `battle.db` mit der Datenbanktabelle `game` erfolgreich eingebunden.

4.3.3 Der Spielstart

Auch an dieser Stelle machen wir uns eine Sicherungskopie der `version_02` indem wir den Ordner kopieren und in `version_03` umbenennen. Den Ordner `version_03` ziehen wir wieder in VSCode. Aber Achtung – für die Datenbank bedeutet das jetzt, dass wir eine neue Kopie angelegt haben, die nichts mit der in der `version_02` zu tun hat. Ihr müsst also bei jeden Versionswechsel auch die neue Datenbank und die neue Tabelle einbinden.

Wie geht es jetzt weiter?

Ich habe für das gesamte Spiel 5 HTML-Seiten vorgesehen. Neben der `base.html` haben wir ja schon die `index.html` als Startseite. Dort befindet sich der Go-Button, wenn wir daraufklicken, geht eine Setup-Seite (`setup.html`) auf, in der sich die beiden Spieler anmelden, der erste Spiel startet das Spiel, generiert einen Code, den er dann dem 2. Spieler mitteilen kann. Spieler 2 meldet sich dann mit diesem Code an und beiden Spielern wird dann die eigentliche Spiel-HTML-Seite (`game.html`) angezeigt.

Wenn das Spiel vorbei ist, gibt es noch eine gesonderte Seite, die das Spielergebnis (`gameOver.html`) beinhaltet.

Wenden wir uns zuerst der `setup.html`-Seite zu. Zum Start soll uns folgender Code reichen:

```
{% extends "base.html" %}
{% block content %}
  <h3>Vorbereitung des Spiels</h3>
{% endblock %}
```

Die Seite soll aus der `index.html` aufgerufen werden, wenn wir den Go-Button klicken. Daher müssen wir die `index.html` wie folgt erweitern:

```
{% extends 'base.html' %} {% block content %}
<form action="{{ url_for('main.setup') }}">
    <h3>Kann losgehen?</h3>
    <div class="button-text-container">
        <button type="submit" class="button-index">Go</button>
    </div>
</form>
{% endblock %}
```

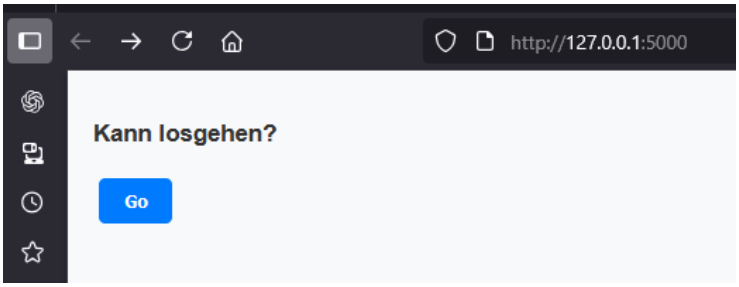
Um den Go-Button haben wir jetzt ein `form`-Tag eingebaut, das einen Verweis auf `main.setup` liefert. Bei Klick auf den Button soll also die Seite http://127.0.0.1:5000/setup aufgerufen werden.

Würden wir zu diesem Zeitpunkt schon die Anwendung neu starten, würden wir einen Fehler bekommen, da `main.setup` nicht erreichbar ist. Die Definition der Seite erfolgt in der `routes.py`. Diese ändern wir wie folgt:

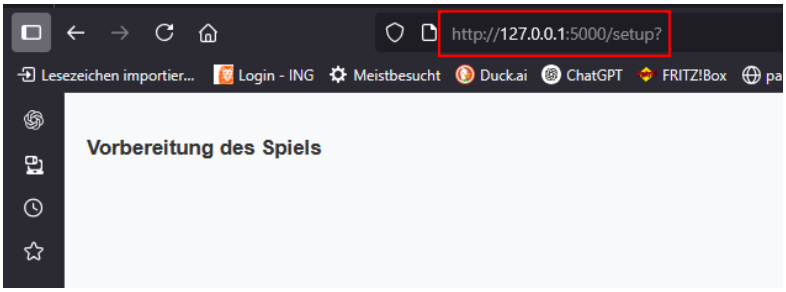
```
from flask import Blueprint, render_template

# Erstelle ein Blueprint für die Routen
main = Blueprint('main', __name__)

@main.route('/')
def index():
    return render_template('index.html')

@main.route("/setup", methods=["GET", "POST"])
def setup():
    return render_template("setup.html")
```

Die 3 letzten Zeilen sind neu, hier wird festgelegt, wann immer `…/setup` aufgerufen wird, wird die Funktion `setup()` ausgeführt, die dann Flask anweist, den Inhalt von `setup.html` zu rendern, also anzuzeigen. `setup` kann Daten erhalten (mittels „`GET`“) und selbst Daten übergeben („`POST`“).

Testen wir das und starten die Anwendung neu. Im Browser erscheint:



Bei Klick auf Go öffnet sich die neue Seite setup.html:



Okay, damit wissen wir, wie der Aufruf einer neuen Seite zu erfolgen hat.

Erweitern wir zunächst unsere `setup.html`. Der neue Code sieht jetzt so aus:

```
Zeile  Inhalt
     1 {% extends "base.html" %}
     2 {% block content %}
     3 <form id="setup-form" method="post">
     4   <h3>Vorbereitung des Spiels</h3>
     5   <p>Entscheide Dich:</p>
     6   <p>Entweder Du bist <b>Spieler 1</b> und startest das Spiel</p>
     7
     8   <!-- Spieler 1 -->
     9   <div>
    10     <label for="p1">Name Spieler 1:</label>
    11     <input id="p1" type="text" name="p_1_name" placeholder="Name" value="{{ p_1_name
       }}"class="fancy-input">
    12   </div>
    13   <div>
    14     <button type="submit" name="p1_pressed" id="p1-btn" value="True" class="button-
       setup">Spieler 1</button>
    15   </div>
    16
    17   <p>Oder Du bist <b>Spieler 2</b> und hast einen Code bekommen</p>
    18   <!-- Spieler 2 -->
    19   <div>
    20     <label for="p2">Name Spieler 2:</label>
    21     <input id="p2" type="text" name="p_2_name" placeholder="Name" value="{{ p_2_name
       }}"class="fancy-input">
    22     <label for="code">Spielcode:</label>
    23     <input id="code" type="text" name="game_code" placeholder="Code" value="{{ game_code
       }}"class="fancy-input">
    24   </div>
    25   <div>
    26     <button type="submit" name="p2_pressed" id="p2-btn" value="True" class="button-
       setup">Spieler 2</button>
    27   </div>
    28
    29 </form>
```

```
30
31 <!-- JavaScript: Pflichtangaben prüfen  -->
32 <script>
33 document.addEventListener("DOMContentLoaded", () => {
34   const form     = document.getElementById("setup-form");
35   const p1Btn    = document.getElementById("p1-btn");
36   const p2Btn    = document.getElementById("p2-btn");
37
38   const p1Name   = form.querySelector('[name="p_1_name"]');
39   const p2Name   = form.querySelector('[name="p_2_name"]');
40   const gameCode = form.querySelector('[name="game_code"]');
41
42   p1Btn.addEventListener("click", () => {
43     // nur Spieler-1-Pflichtfeld aktivieren
44     p1Name.required   = true;
45     p2Name.required   = false;
46     gameCode.required = false;
47   });
48
49   p2Btn.addEventListener("click", () => {
50     // nur Spieler-2-Pflichtfelder aktivieren
51     p1Name.required   = false;
52     p2Name.required   = true;
53     gameCode.required = true;
54   });
55 });
56 </script>
57 {% endblock %}
```

Wie Ihr seht hier mal wieder die Darstellung in einer Tabelle, da ich auf bestimmte Zeilennummern eingehen möchte.

Bevor Ihr das ausführt, bitte die zugehörigen „fancy-input“-Einstellungen in der `style.css` hinterlegen:

```
/* Input-Felder */
.fancy-input {
  width: 250px;
  margin-left: 10px;
  margin-right: 10px;
  padding: 10px 14px;
  font-size: 16px;
  border: 2px solid #ccc;
  border-radius: 5px;
  background-color: #f9f9f9;
  color: #333;
  outline: none;
  transition: border-color 0.3s, box-shadow 0.3s;
  box-shadow: inset 0 1px 3px rgba(0, 0, 0, 0.1);
}

.fancy-input:focus {
```

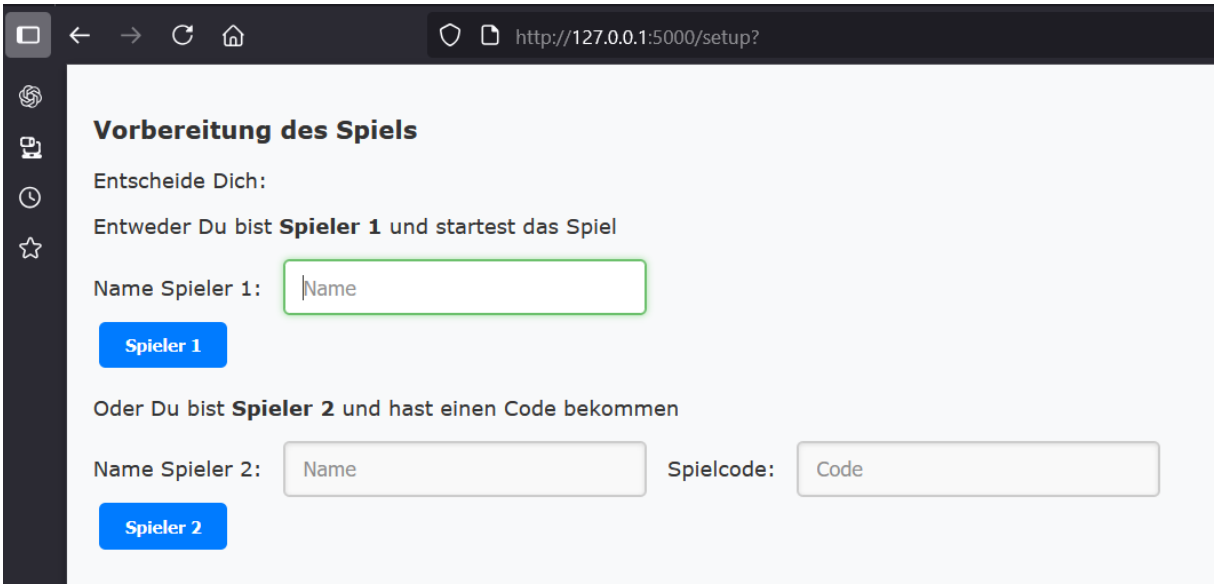
```
  border-color: #6cc06c;              /* grüner Fokusrahmen */
  box-shadow: 0 0 6px #a2e2a2;
  background-color: #ffffff;
}
```

Und damit die beiden Button so aussehen wie in der `index.html` noch die Button um eine Zeile erweitern:

```
/* Button */
.button-index,
.button-setup {
  padding: 10px 20px;
```

Spielt da unbedingt mit den Einstellungen rum!

Nach Neustart des Servers (Anhalten des Servers mit „STRG+C“ im Terminalfenster, dann wieder „`flask run`“) solltet Ihr bei Klick auf den Go-Button die neue Seite `setup.html` angezeigt bekommen:



Damit erklärt sich schon vieles im Code, oder?

Wir haben ein großes Formular (Zeilen 3 bis 29 in obiger Tabelle), ab Zeile 32 stehen die JavaScript-Prüfungen, darauf komme ich gleich.

Ab Zeile 8 ist die Eingabe um den Spieler 1 definiert, ab Zeile 19 dann für den Spieler 2. Spieler 1 hat nur den Namen, den er für dieses Spiel benutzen möchte, Spieler 2 hat zusätzlich noch ein Eingabefeld für den Spielcode, den ihm Spieler 1 zurufen muss.

Damit sind wir dann schon beim Teil mit dem JavaScript-Code. Mit „`DOMContentLoaded`“ greifen wir auf das „Data Object Model“ (DOM) zu. Das stellt sicher, dass wir Informationen erst dann abfragen, wenn das Dokument komplett geladen ist.

Mit dem Zugriff auf `x.required = true`, sagen wir, dass das Feld erforderlich ist, sonst geht es nicht weiter. Die Fehlermeldung, die angezeigt wird, wenn das Feld leer ist und trotzdem auf einen Button geklickt wird, stammt direkt aus den Browser-Einstellungen und müssen nicht extra definiert werden.

Jetzt müssen wir nur noch das `routes.py`-Modul anpassen, um die Einträge in `setup.html` verarbeiten zu können. Da wir uns da etwas länger aufhalten, spendiere ich mal ein neues Kapitel, wir bleiben aber noch in der `version_03`.

4.3.4 Der Spielcode

Das Modul `routes.py` stellt unsere Server- oder „Backend“-Verarbeitung dar. Hier nehmen wir die Informationen aus den verschiedenen Clients oder „Frontends“ – also den HTML-Seiten – entgegen, vergleichen, ziehen Schlüsse und leiten Handlungen ab.

Was müssen wir in dieser Phase des Spiels tun?

Zuerst kümmern wir uns um die Erstellung und Speicherung eines neuen Spiel-Codes, wenn Spieler 1 seinen Namen eingegeben und den Spieler-1-Button geklickt hat.

Erstellen wir ein neues Modul `generate_code.py`. Ich habe da schonmal was vorbereitet:

```
Zeile  Inhalt
     1 import random
     2 from string import ascii_uppercase
     3 from app_init import db
     4 from sqlalchemy import func
     5 from models import Game
     6
     7 # Funktion liefert Großbuchstaben für die übergebene
     8 # Anzahl an Stellen
     9 def gen(length):
    10     gen_code = ""
    11     for _ in range(length):
    12         gen_code += random.choice(ascii_uppercase)
    13
    14     return gen_code
    15
    16 # Funktion schaut nach, ob der generierte Code schon
    17 # vorhanden ist und liefert True Oder False zurück
    18 def lookup(gen_code):
    19     valid = True
    20     lookup = Game.query.all()
    21
    22     for item in lookup:
    23         if str(item) == gen_code:
    24             valid = False
    25             break
    26
    27     return valid
    28
    29 #Funktion liefert den generierten Code zurück
    30 def generate_unique_code(length):
    31     generate_new = True
    32     is_valid = False
    33     gen_code = ""
    34     # Testschalter - falls gewünscht, unten einen
    35     # in der DB existierenden Code mitgeben und
```

```
36      # Schalter auf True setzen
37      test = False
38
39      while generate_new:
40          is_valid = False
41          while not is_valid:
42              if test:
43                  #hier einen existierenden Code eingeben
44                  gen_code = "ABCDEF"
45                  test = False
46              else:
47                  gen_code = gen(length)
48
49              is_valid = lookup(gen_code)
50
51              if is_valid:
52                  generate_new = False
53              else:
54                  gen_code = gen(length)
55                  is_valid = lookup(gen_code)
56
57          generate_new = False
58
59      return gen_code
```

Hier haben wir 3 Funktionen zusammengesteckt. Von außen, also aus unserer `routes.py` wird die Funktion `generate_unique_code(length)` aufgerufen. Sie erwartet eine Längenangabe für die Anzahl Stellen des zu generierenden Codes und gibt den fertigen Code in Zeile 59 zurück. Ich hätte auch direkt 6 Stellen vorgeben können, vielleicht wollt Ihr aber erstmal mit 4 Stellen starten.

Die Funktion `gen(length)` wird erstmals in Zeile 47 aufgerufen, sie erstellt einen neuen Code, bestehend aus einer Menge zufällig ausgewählten Großbuchstaben. Die Anzahl wird wie beschrieben von außen mitgegeben.

Die Funktion `lookup(gen_code)` wird mit dem frisch erstellen Code aufgerufen. Mit `Game.query.all()` werden alle existierenden Einträge in der Datenbank gelesen. Sollte der Code bereits vorhanden sein, wird der Booleanwert `valid` auf `False` gesetzt andernfalls bleibt er auf der Voreinstellung `True` und wird der rufenden Funktion zurückgegeben.

Falls `valid` also `False` sein sollte, wird ein neuer Code solange generiert und geprüft, bis `valid` den Wert `True` hat. Dann wird der neue, validierte, also noch nicht in der Datenbank enthaltene Code zurückgegeben.

Bauen wir die Funktion in die `routes.py` ein. Zuerst die Erweiterung der import-Anweisungen um das Modul `request` aus der Flask-Bibliothek und unser neues Modul:

```
from flask import Blueprint, render_template, request
import generate_code
```

Mit `request` haben wir Zugriff auf die Daten, die uns in der HTML-Seite übermittelt werden. Der Ausschnitt für die `setup`-Route sieht jetzt so aus:

```
@main.route("/setup", methods=["GET", "POST"])
def setup():

    # Wenn Methode "POST" ist, erwarten wir Formulardaten
    if request.method == "POST":
        if request.form.get('p1_pressed') == 'True':
            p_1_name = request.form.get("p_1_name", "").strip()
            print("Button für Spieler 1 wurde gedrückt")
            # Generiere einen eindeutigen Code
            game_code = generate_code.generate_unique_code(6)
            print(f"Generierter Code: {game_code} und Spieler 1 Name: {p_1_name}")
            return render_template(
                "setup.html"
            )
        else:
            return render_template(
                "setup.html"
            )
    else:
        return render_template(
            "setup.html"
        )
```

Das ist selbsterklärend, oder? Damit nichts schief geht, geben wir immer `setup.html` in die Aufbereitung. Nur, wenn wir auf den „Spieler 1“-Button geklickt haben, rufen wir die Funktion `generate_unique_code()` mit der Anzahl Stellen von „6“ auf.

Vor dem nächsten Test bitte unbedingt den Server wieder neu starten. Dann solltet auch Ihr so einen Eintrag im Terminalfenster sehen:

Bei Euch auch? Super, dann erweitern wir die `setup`-Route jetzt um das Speichern des neuen Spiels in der Datenbanktabelle. Der neue Ausschnitt sieht so aus:

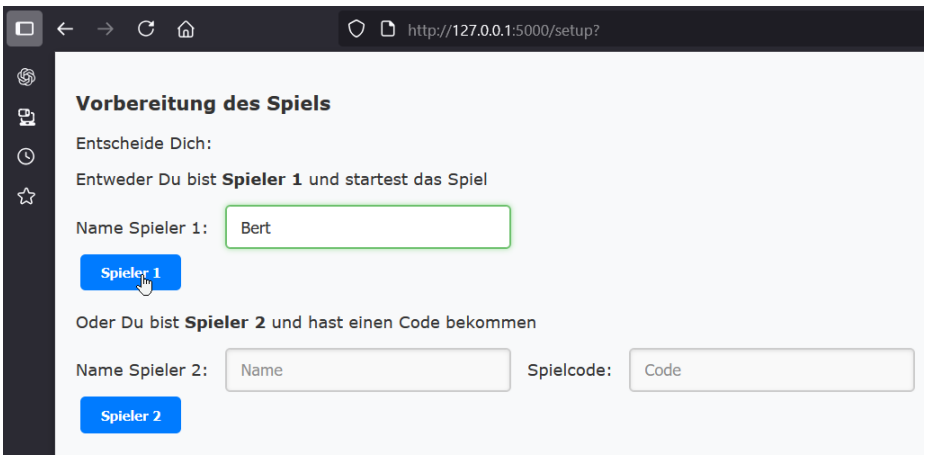
```
            …
            # Generiere einen eindeutigen Code
            game_code = generate_code.generate_unique_code(6)
            print(f"Generierter Code: {game_code} und Spieler 1 Name: {p_1_name}")
            new_game = Game(game_code=game_code,
                            p_1_name=p_1_name,
                            p_1_status="Spiel generiert")
            db.session.add(new_game)
            db.session.commit()
            return render_template(
                "setup.html"
            )
            …
```

Erweiterung in den Importen:

```
from flask import Blueprint, render_template, request
import generate_code
from app_init import db
from models import Game
```

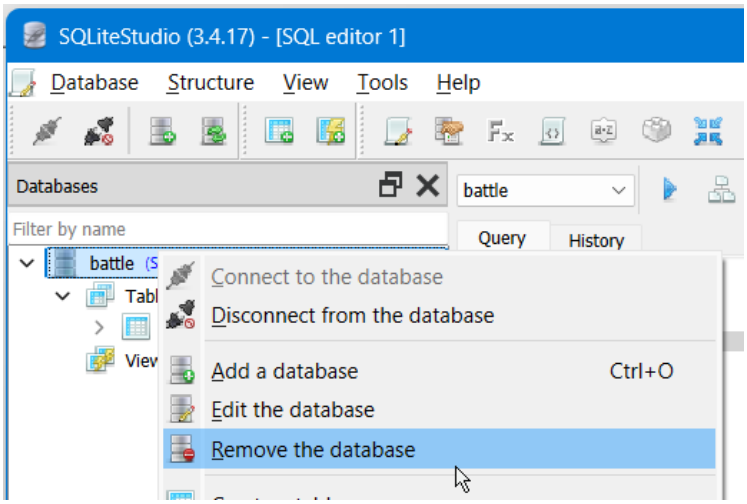
Wenn wir das jetzt wieder Testen (Neustart Server(!)), sollten wir in der Datenbanktabelle jetzt einen neuen Eintrag haben.

Eingabe im Frontend:

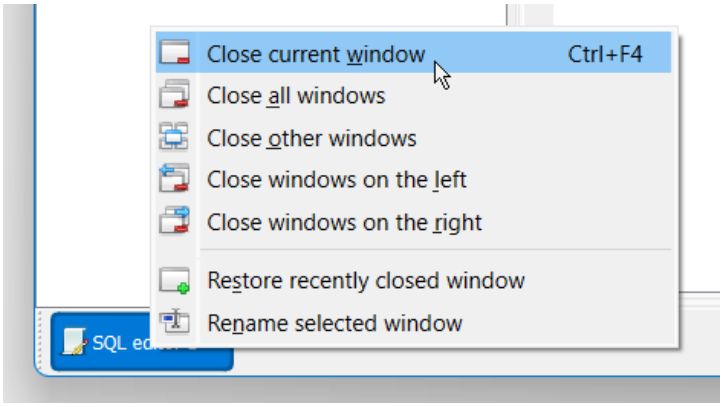


Terminalfenster gibt aus:

Blicken wir in die Datenbank. Wenn bei Euch noch die Instanz aus `version_02` angezeigt wird, müsst Ihr die neue Instanz von `version_03` einbinden.

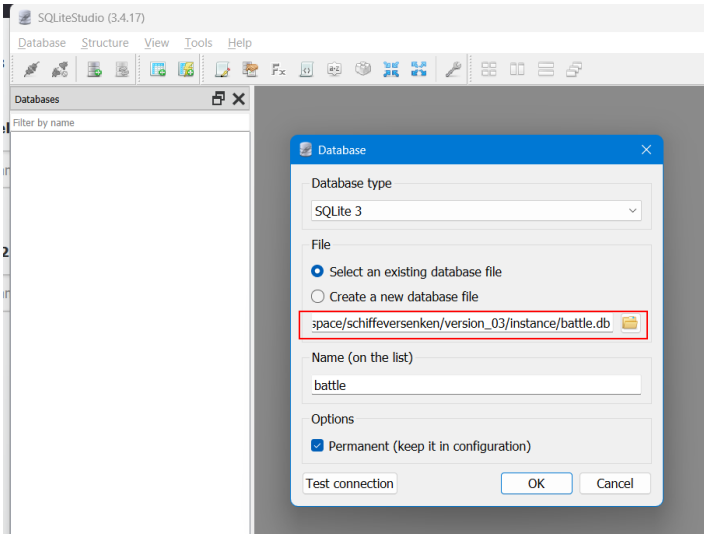


Dazu mit Rechtklick auf die alte Instanz das Kontext-Menu aufrufen und „Remove the database“ anklicken.



Bevor es Verwirrung gibt, auch den Texteditor unten links zumachen.

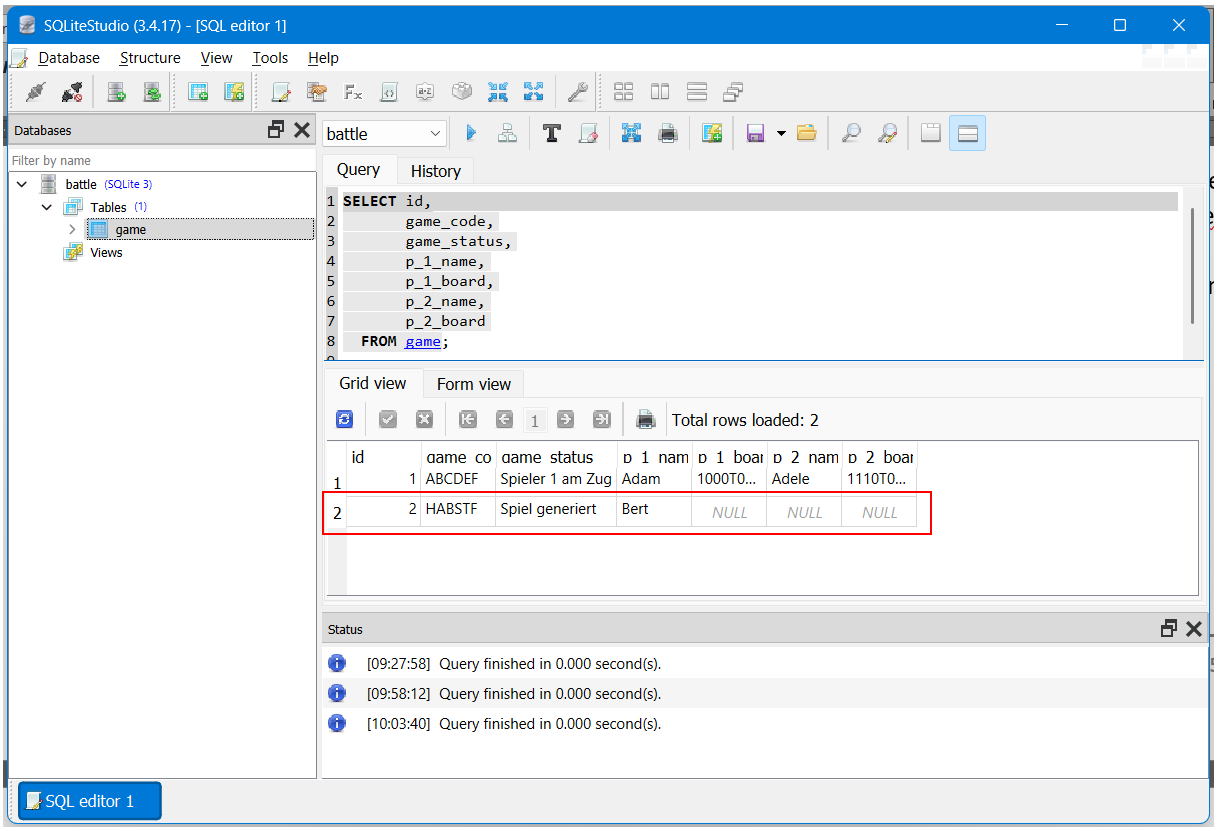
Dann wie am Anfang beschrieben, die neue Instanz in `version_03` auswählen.



Die Reihenfolge war:

- Oben Database/Add a database
- Neue Instanz im Explorer auswählen
- Rechtsklick auf `battle.db`, dann Connect to the database
- Rechtsklick auf Datenbanktabelle `game`, dann `Generate query for table`, dann `SELECT`
- Im rechten Fenster oben auf den blauen Pfeil klicken.

Die Ausgabe zeigt bei mir:



Yes, alles richtig gemacht. Der zweite Datensatz wurde erfolgreich angelegt!

Wenden wir uns nun Spieler 2 zu.

Wenn Spieler 1 das Spiel erzeugt hat, muss der Code irgendwie zu Spieler 2 gelangen. Ich gehe jetzt mal davon aus, dass beide im gleichen Raum sind, oder per Telefon verbunden, also Spieler 1 hat auf jeden Fall die Möglichkeit Spieler 2 den Code zu übermitteln.

Wenn also Spieler 2 sich anmeldet, gibt er den Namen und den Spiel-Code an. Wir prüfen, ob das angegebene Spiel in der Datenbank ist und den Status „`Spiel generiert`“ hat. Dann können wir einigermaßen sicher sein, dass Spieler 2 auf dem richtigen Weg ist. Sollte beides passen, müssen wir den Datensatz mit dem Namen von Spieler 2 aktualisieren und den Status ändern, nehmen wir für diese Phase des Spiels den Status „`Spiel gestartet`“.

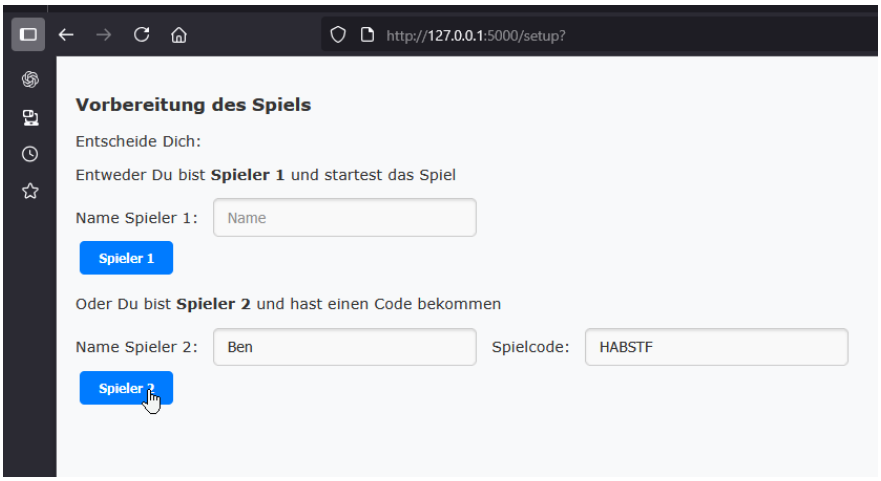
Die neue Route sieht dann so aus:

```
@main.route("/setup", methods=["GET", "POST"])
def setup():

    # Wenn Methode "POST" ist, erwarten wir Formulardaten
    if request.method == "POST":
        if request.form.get('p1_pressed') == 'True':
            …
            return render_template(
                "setup.html"
            )
        # hier könnten wir auch else: setzen, da wir nur bei Klick auf einen
        # der beiden Button hier landen
        if request.form.get('p2_pressed') == 'True':
            p_2_name = request.form.get("p_2_name", "").strip()
```

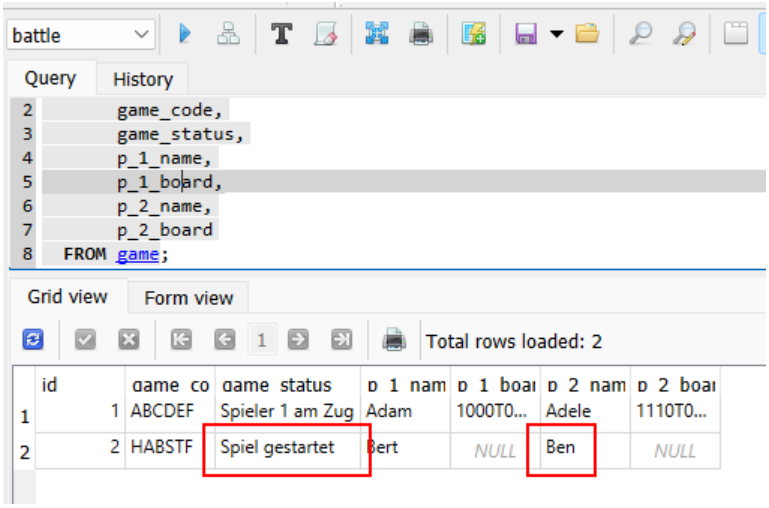
```
            game_code = request.form.get("game_code", "").strip()
            #Versuch, den Datensatz mit dem angegebenen Code zu finden
            existing_game = Game.query.filter_by(game_code=game_code).first()
            if not existing_game:
                print(f"Ups {p_2_name}, der Code {game_code} existiert nicht")
            else:
                existing_game.p_2_name = p_2_name
                existing_game.game_status = "Spiel gestartet"
                db.session.commit()
            return render_template(
                "setup.html"
            )
    else:
        return render_template(
            "setup.html"
        )
```

Das ist nur ein Übergang, um das Update auf der Datenbank zu testen, gleich machen wir die gesamte Route schick. Jetzt aber erst einmal einen Neustart!

Geben wir also Spieler 2 den Namen „Ben“, setzen den Code auf „HABSTF“ und drücken den Spieler 2-Button sollte das Update in der Datenbanktabelle angekommen sein:



Kontrolle in der DB:



Super, auch das hat geklappt

Geben wir einen Code für Spieler 2 mit, der *nicht* in den Datenbanktabelle gespeichert ist, sollte im Terminal-Fenster der entsprechende Hinweis auftauchen:

```
Ups Foo, der Code AAAAAA existiert nicht
```

Das ist allerdings noch keine Ausgabe auf dem Frontend, Spieler 2 hat also noch keine Informationen darüber, dass das Speichern nicht geklappt hat.

Wir können eine Fehlermeldung mit der `error`-Funktion auf der Seite sichtbar machen. Dazu müssen wir aber an 4 Stellen ran:

1 – Zu Beginn der Route müssen wir `error` initialisieren:

```
@main.route("/setup", methods=["GET", "POST"])
def setup():
    error = None
    # Wenn Methode "POST" ist, erwarten wir Formulardaten
    if request.method == "POST":
        …
```

2 – weiter unten ersetzen wir die Zeile mit der `print`-Anweisung und erweitern den Rücksprung auf `setup.html`:

```
        …
        if request.form.get('p2_pressed') == 'True':
            p_2_name = request.form.get("p_2_name", "").strip()
            game_code = request.form.get("game_code", "").strip()
            #Versuch, den Datensatz mit dem angegebenen Code zu finden
            existing_game = Game.query.filter_by(game_code=game_code).first()
            if not existing_game:
                error = f"Ups {p_2_name}, der Code {game_code} existiert nicht"
                return render_template(
                    "setup.html",
                    error=error,
                    p_2_name=p_2_name,
                    game_code=game_code
                )
            else:
                …
```

3 – Damit wir das dem Spieler anzeigen, muss natürlich auch `setup.html` selbst angepasst werden. Ganz am Ende, vor dem schließenden `form`-Tag fügen wir die grün geschriebenen Zeilen ein:

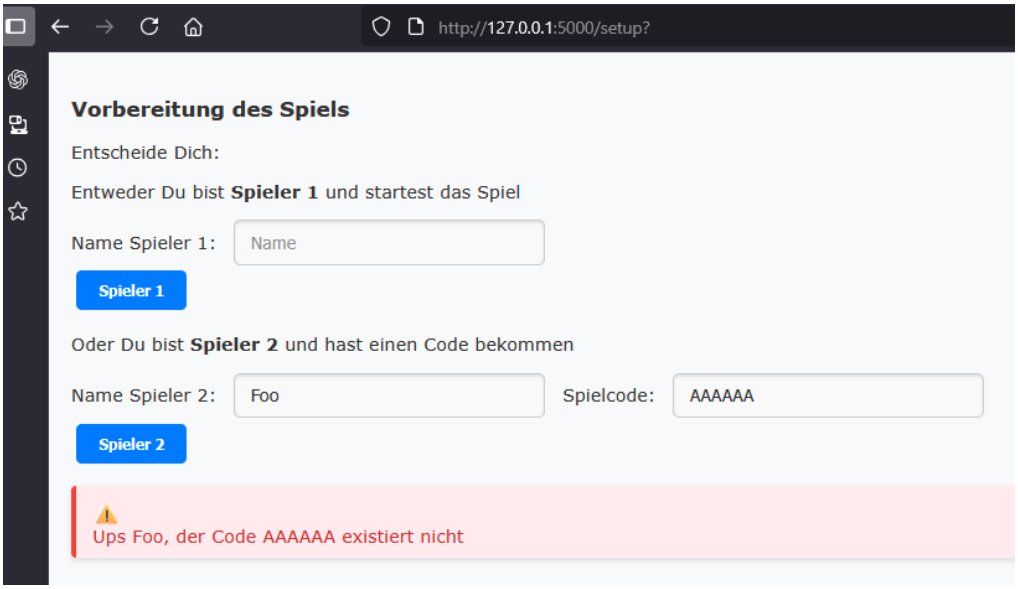
```
<form id="setup-form" method="post">
…

  {% if error %}
  <ul class="error">
    <li>{{ error }}</li>
  </ul>
  {% endif %}
</form>
```

4 – die Klasse „`error`“ müssen wir natürlich noch in der `style.css` aufnehmen, ans Ende kopieren wir:

```
.error {
    background-color: #ffebee;
    border-left: 5px solid #f44336;
    color: #d32f2f;
    padding: 10px 15px;
    margin: 15px 0;
    border-radius: 4px;
```

```
    font-weight: 500;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
    list-style-type: none;
    animation: fadeIn 0.3s ease-in-out;
    position: relative;
}

.error::before {
    content: '⚠';
    margin-right: 10px;
    font-size: 1.2em;
}

@keyframes fadeIn {
    from { opacity: 0; transform: translateY(-10px); }
    to { opacity: 1; transform: translateY(0); }
}
```

Auf der `setup`-Seite sieht das dann (nach Neustart) so aus:



Sehr gut, jetzt bleibt noch die Weiterleitung an die eigentliche Spiel-Seite `game.html`.

Wir erstellen im Ordner `templates` die neue Datei `game.html`. Für einen ersten Wurf füllen wir sie mit folgendem Inhalt:

```
{% extends 'base.html' %}
{% block content %}

<div>
  <h3>aktueller Spieler ist {{ p_actual }}, Spielcode ist {{ game_code }}
</h3>
</div>

{% endblock %}
```

Das kennen wir ja schon, wir nutzen zur Kontrolle hier wieder Jinja2-Code. Um `p_actual` kümmern wir uns gleich.

Wann rufen wir `game.html` auf?

Gehen wir noch einmal die `setup`-Route durch. Der Aufruf erfolgt

- Entweder, wir sind Spieler 1 und haben erfolgreich ein neues Spiel generiert,
- oder wir sind Spieler 2 und haben einen existierenden Code eingegeben

Da fällt mir auf, wir haben die Prüfung auf den Status „`Spiel generiert`“ noch nicht eingebaut, das erledigen wir gleich mit.

Damit ich wieder Zeilennummern nennen kann, hier die finale Route:

```
Zeile Inhalt
    1 @main.route("/setup", methods=["GET", "POST"])
    2 def setup():
    3     error = None
    4     board_init = "0000000000" * 10  # 10x10 Spielfeld, initial leer
    5     # Wenn Methode "POST" ist, erwarten wir Formulardaten
    6     if request.method == "POST":
    7         if request.form.get('p1_pressed') == 'True':
    8             p_1_name = request.form.get("p_1_name", "").strip()
    9             # Generiere einen eindeutigen Code
   10             game_code = generate_code.generate_unique_code(6)
   11             new_game = Game(game_code=game_code,
   12                             p_1_name=p_1_name,
   13                             p_1_board=board_init,
   14                             p_1_status="Spiel generiert")
   15             db.session.add(new_game)
   16             db.session.commit()
   17             session["p_actual"] = p_1_name
   18             return redirect(url_for("main.game", game_code=game_code))
   19
   20         # hier könnten wir auch else: setzen, da wir nur bei Klick auf einen
   21         # der beiden Button hier landen
   22         if request.form.get('p2_pressed') == 'True':
   23             p_2_name = request.form.get("p_2_name", "").strip()
   24             game_code = request.form.get("game_code", "").strip()
   25             #Versuch, den Datensatz mit dem angegebenen Code zu finden
   26             existing_game = Game.query.filter_by(game_code=game_code).first()
   27             if not existing_game:
   28                 error = f"Ups {p_2_name}, der Code {game_code} existiert nicht"
   29                 return render_template(
   30                     "setup.html",
   31                     error=error,
   32                     p_2_name=p_2_name,
   33                     game_code=game_code
   34                 )
   35             else:
   36                 if existing_game.p_2_name != None:
   37                     error = f"Ups {p_2_name}, das Spiel {game_code} hat schon 2
      Spieler"
   38                     return render_template(
   39                         "setup.html",
   40                         error=error,
   41                         p_2_name=p_2_name,
```

```
42                          game_code=game_code
43                      )
44                  else:
45                      p_1_name = existing_game.p_1_name
46                      game_code = existing_game.game_code
47                      existing_game.p_2_name = p_2_name
48                      existing_game.p_2_board = board_init
49                      existing_game.p_2_status = "Spieler 2 angemeldet"
50                      db.session.commit()
51                      session["p_actual"] = p_2_name
52                      return redirect(url_for("main.game", game_code=game_code))
53
54      else:
55          return render_template(
56              "setup.html"
57          )
```

Bitte noch die Erweiterung des Imports machen:

```
from flask import Blueprint, render_template, request, redirect, url_for, session
```

In Zeile 4 erstellen wir ein initiales Spielfeld, das wir in Zeile 13 Spieler 1 mitgeben und in Zeile 48 Spieler 2.

In Zeile 17 öffnen wir eine `session`, das ist ähnlich wie ein Cookey, hier übergeben wir den Namen von Spieler 1 als aktuellem Spieler. Das brauchen wir später, um unterscheiden zu können, wer gerade `games.html` nutzt.

In Zeile 18 rufen wir `game.html` mit dem Inhalt von `game_code` per `redirekt` auf. Damit verlassen wir `setup.html`.

Zeile 51 übergeben wir den Namen von Spieler 2 in die `session`-Variable `p_actual`.

Stand Zeile 52 ist:

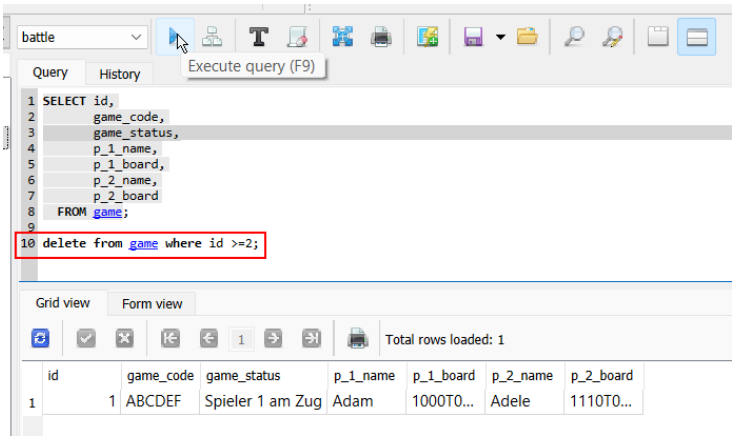
- wir haben geprüft, dass der Button Spieler 2 gedrückt wurde (Abfrage Zeile 22 – „`if request.form.get('p2_pressed') == 'True':`“),
- der Code ist gefunden worden (wir sind im `else`-Zweig Zeile 35 der Abfrage „`if not existing_game:`“ Zeile 27),
- der Status für Spieler 2 ist „Spieler 2 angemeldet“, (wir befinden uns im `else`-Zweig Zeile 44 von „`if existing_game.game_status != None:`“ aus Zeile 36)

Jetzt brauchen wir noch eine minimale Route für `game`, sonst funktioniert der `redirekt` nicht:

```
@main.route("/game/<game_code>")
def game(game_code):
    game_code=game_code
    p_actual = session.get("p_actual")

    return render_template(
        "game.html",
        game_code=game_code,
        p_actual=p_actual
    )
```

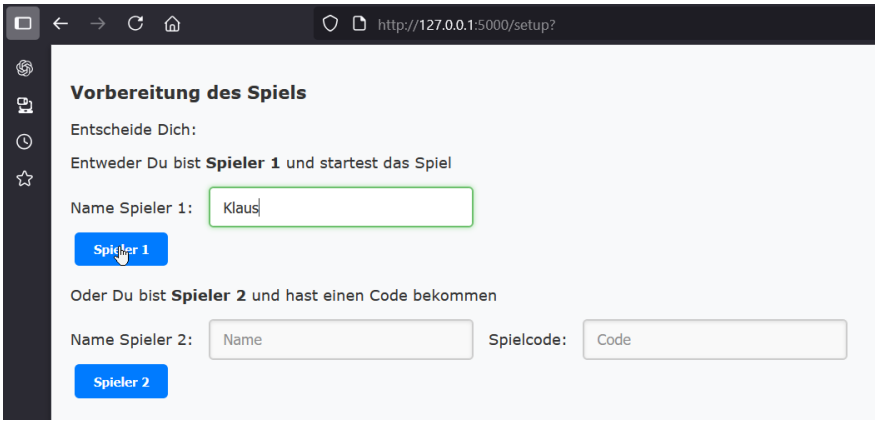
Das sollte es mit dem Setup des Spiels gewesen sein. Wir testen das nach Restart des Servers.

Aktueller Stand in der Datenbanktabelle ist:

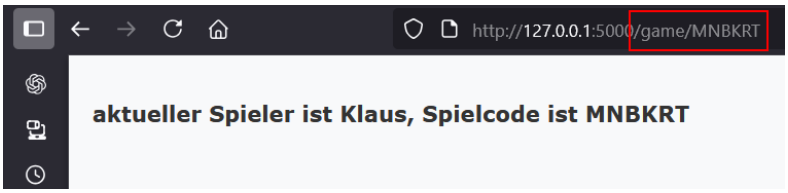


Da sich in der Zwischenzeit eine große Menge an Einträgen angesammelt hat, habe ich alles bis auf den ersten Datensatz gelöscht. Der Befehl dazu findet sich in Zeile 10: „`delete from game where id >=2;`“. Achtung, zum Ausführen muss sich der Cursor im oberen Fenster irgendwo in Zeile 9 oder 10 befinden, auf jeden Fall *nach* dem Semikolon des Select-Befehls und *vor* dem Semikolon des `delete`-Befehls.

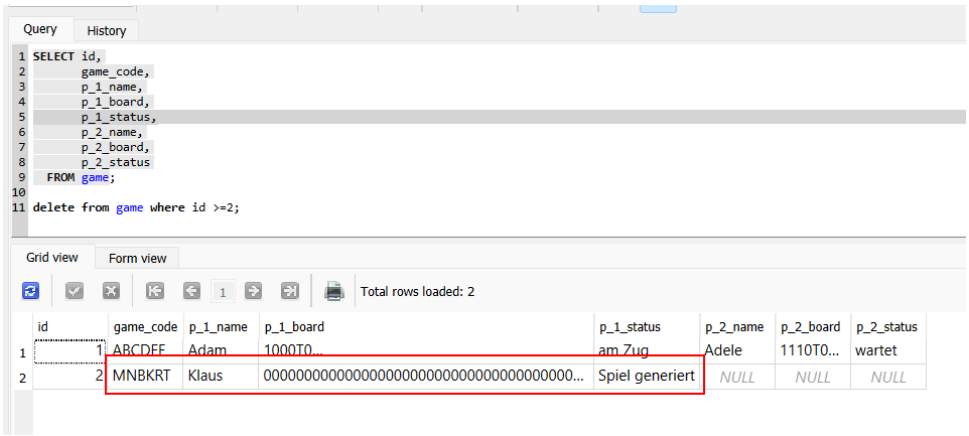
Zurück zum Spiel, Spieler Klaus eröffnet das Spiel:



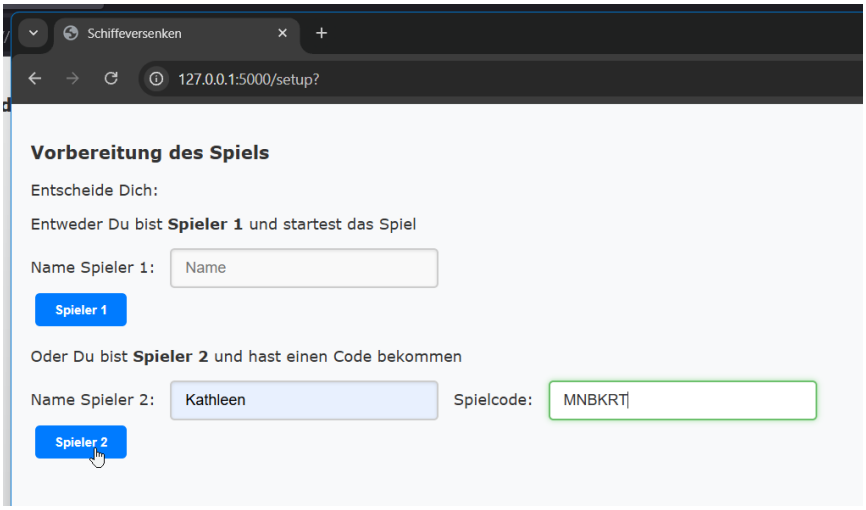
Sprung auf `game.html`, der `redirekt` hat geklappt:



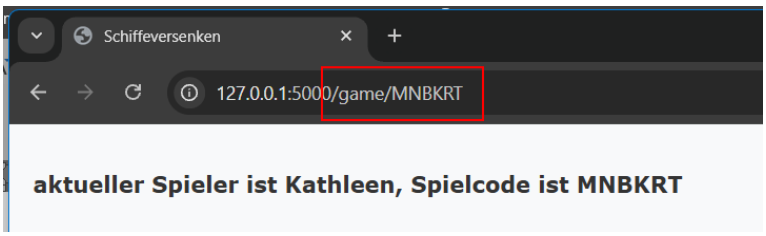
Der Eintrag in der Datenbanktabelle hat auch funktioniert:



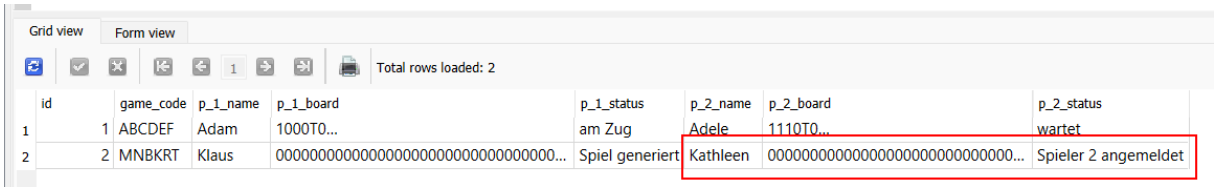
Kathleen kommt dazu. Wir öffnen dazu **den anderen Browser**, geben http://127.0.0.1:5000 in die Suchleiste ein, dann Go und landen auf unserer `setup.html`:



Der `redirekt` auf `game.html` hat auch hier funktioniert:



Auch hier wieder die Kontrolle in der Datenbanktabelle:



Super, das sieht doch gut aus.

Ich hoffe, dass das bei Euch so aussieht, falls nicht, ladet Euch die `version_03` in der Projektseite auf meiner Homepage runter, entpackt das File und zieht es in Euren Code-Editor. Dann solltet Ihr auf dem gleichen Stand sein.

Für die weiteren Schritte erstellen wir daher wieder eine neue Kopie und arbeiten mit `version_04` weiter.

4.3.5 Die Spielfelder

An dieser Stelle machen wir uns wieder Gedanken über die nächsten Schritte. Was wissen wir über das Spiel? Wie gestalten wir die Spielfelder? Wie werden die Schiffe platziert? Wie erkennen wir, wer am Zug ist?

Fangen wir mit den Spielfeldern an. Die Anforderung war, 2 Spielfelder nebeneinander anzuzeigen, das eigene Spielfeld und das des Gegners. Die Größe wurde mit „10 mal 10 Kästchen“ festgelegt. Da wir kein Karo-Papier haben, müssen wir uns die Kästchen selber erstellen.

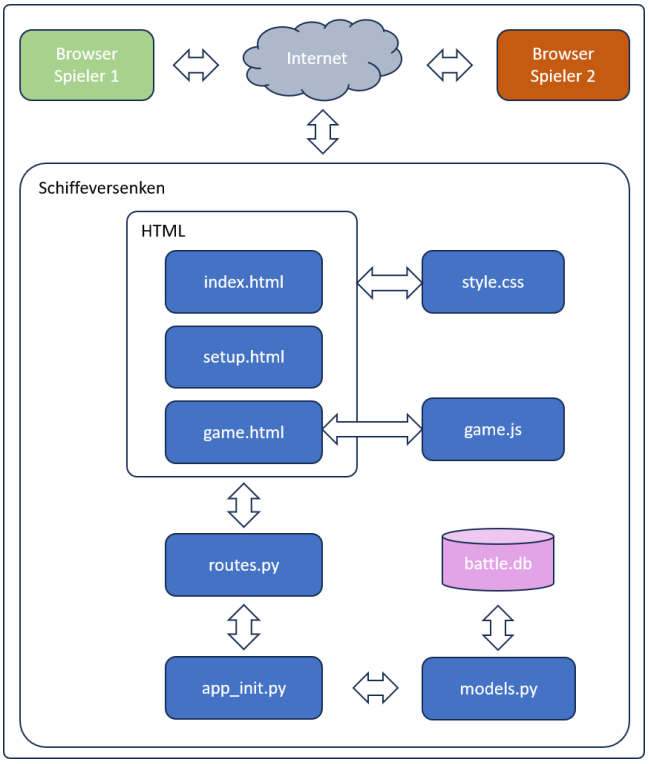
Damit wir auch echte Koordinaten haben, sollten wir in der ersten Zeile noch die Buchstaben A bis J ausgeben, und die Spalten von 1 bis 10 durchnummerieren.

Gehen wir das zuerst an.

Da wir jetzt viel mit Aussehen und Darstellung von Inhalten der `game.html` zu tun haben, kommt jetzt auch JavaScript zum Einsatz. Die Verbindung zwischen der HTML-Datei und JavaScript-Dokument erfolgt im unteren Teil der `game.html`:

```
    …
    </div>
</div>
<script src="{{ url_for('static', filename='js/game.js') }}"></script>
…
{% endblock %}
```

Da wir jetzt mit der game.js auch JavaScript an Bord nehmen, sollten wir uns vielleicht die Interaktion aller beteiligten Komponenten mal genauer anschauen. Gehen wir es an einem Bild durch.



Gehen wir es kurz durch.

Beide Spieler melden sich jeweils über Ihre Browser via Internet an unserer Applikation „Schiffeversenken“ an.

Start ist die HTML-Seite `index.html`. Im Wechsel mit der Python-Datei `routes.py` erfolgt die Verzweigung innerhalb der Anwendung. In der HTML-Seite `setup.html` erfolgt die Anmeldung, entweder als Spieler 1 oder als Spieler 2.

Hat die Anmeldung als Spieler 1 oder Spieler 2 geklappt, erfolgt die Weiterleitung an die HTML-Seite `game.html` in der wir bis zum Ende des Spiels bleiben.

Auf alle HTML-Seiten wirkt die CSS-Datei `styles.css`.

Als Unterstützung der `game.html` wird jetzt die JavaScript-Datei `game.js` herangezogen, das lernen wir in diesem Kapitel.

Die Kommunikation mit der Datenbank übernimmt das Python-Modul `app_init.py` in Zusammenarbeit mit `models.py`.

Fein, wir haben also alle Komponenten zusammen. Dann gehen wir jetzt die Erstellung der beiden Spielfelder an.

Ich fange mal mit der `game.html` an. Der erweiterte Code lautet:

```
Zeile  Inhalt
    1  {% extends 'base.html' %}
    2  {% block content %}
    3
    4  <div class="first_row">
    5    <h3>Spieler 1: {{ p_1_name }} - Spieler 2: {{ p_2_name }} - Spielcode ist
       {{ game_code }}</h3>
    6  </div>
    7
    8  <div class="game-board">
    9    <!-- Eigenes Feld -->
   10    <div class="board-container">
   11      <h3>Eigenes Feld ({{ p_actual }})</h3>
   12      <div class="board-wrapper">
   13        <div class="row-labels">
   14          <!-- Wird später mit JS gefüllt: 1 bis 10 -->
   15        </div>
   16        <div class="col-wrapper">
   17          <div class="col-labels">
   18            <!-- Wird später mit JS gefüllt: A bis J -->
   19          </div>
   20          <div id="myBoard" class="board"></div>
   21        </div>
```

```
    </div>

  </div>

  <!-- Gegnerisches Feld -->
  <div class="board-container">
    <h3>Gegner Feld ({{ p_opponent }})</h3>
    <div class="board-wrapper">
      <div class="row-labels">
        <!-- Wird später mit JS gefüllt: 1 bis 10 -->
      </div>
      <div class="col-wrapper">
        <div class="col-labels">
          <!-- Wird später mit JS gefüllt: A bis J -->
        </div>
        <div id="oppBoard" class="board"></div>
      </div>
    </div>
    <div class="status" id="status">Status: Warte auf deinen Zug...</div>
  </div>
</div>

<script src="{{ url_for('static', filename='js/game.js') }}"></script>
<script>
  initGame({
    gameCode: "{{ game_code }}",
    p_actual: "{{ p_actual }}",
    p1:       "{{ p_1_name }}",
    p2:       "{{ p_2_name }}",
    p1Status: "{{ p_1_status }}",
    p2Status: "{{ p_2_status }}",
    myBoard:  "{{ my_board }}",
    oppBoard: "{{ opp_board }}"
  });
</script>
{% endblock %}
```

Zu HTML gehört immer das CSS, wissen wir, die Erweiterung des `style.css` sieht bei mir so aus:

Zeile	Inhalt

```
...
/* Erste Zeile */
.first_row {
  text-align: center;
  margin-bottom: 20px;
}

/* Infoanzeige */
#info {
  margin-top: 20px;
  font-size: 16px;
  font-weight: bold;
  min-height: 1.5em;
```

```
14 }
15
16 /* Gameboard-Container */
17 .game-board {
18   display: flex;
19   justify-content: center;
20   gap: 40px;
21   flex-wrap: wrap;
22   margin-bottom: 20px;
23 }
24
25 /* Einzelnes Board */
26 .board-container {
27   text-align: center;
28 }
29
30 .board {
31   display: grid;
32   grid-template-columns: repeat(10, 40px);
33   grid-template-rows: repeat(10, 40px);
34   gap: 0px;
35   margin-top: 10px;
36   justify-content: center;
37 }
38
39
40 /* Zahlen und Buchstaben */
41 .board-wrapper {
42   display: flex;
43   align-items: start;
44   justify-content: center;
45   gap: 2px;
46 }
47
48 /* Zahlen */
49 .row-labels {
50   display: grid;
51   grid-template-rows: 40px repeat(10, 40px);
52   margin-top: 5px;
53   margin-right: 5px;
54   text-align: right;
55   gap: 0px;
56 }
57
58 .label-spacer {
59   height: 40px;
60   gap: 2px;
61 }
```

```
 62
 63 .col-board {
 64   display: flex;
 65   flex-direction: column;
 66   align-items: center;
 67 }
 68
 69 /* Buchstaben */
 70 .col-labels {
 71   display: grid;
 72   grid-template-columns: repeat(10, 40px);
 73   height: 40px;
 74   margin-bottom: 2px;
 75   padding-bottom: 0px;
 76   gap: 0px;
 77 }
 78
 79 .col-labels div,
 80 .row-labels div {
 81   display: flex;
 82   justify-content: center;
 83   align-items: center;
 84   font-weight: bold;
 85   font-size: 14px;
 86   box-sizing: border-box;
 87   padding-bottom: 1px;
 88 }
 89
 90 .col-labels div{
 91   width: 40px;
 92   height: 60px;
 93 }
 94 .row-labels div {
 95   width: 20px;
 96   height: 40px;
 97 }
 98
 99 /* Zellen */
100 .cell {
101   width: 40px;
102   height: 40px;
103   background-color: #cce5ff;
104   border: 1px solid #99c2ff;
105   box-sizing: border-box;
106   cursor: pointer;
107 }
108
109 .cell.ship {
```

```
110   background-color: #687b91;
111 }
112
113 .cell.hit {
114   background-color: red;
115 }
116
117 .cell.miss {
118   background-color: #5fc1ee;
119   border: 2px dashed #ccc;
120 }
121
122 .cell.opponent-hit {
123   background-color: #d6b3ff;   /* hell violett */
124 }
125
126 .cell.opponent-miss {
127   background-color: var(--blue-200, #5fc1ee);
128 }
```

Ja, das ist ziemlich viel, ich gehe das jetzt auch nicht im Einzelnen durch. Das ist ja alles nur eine Vorlage, wie Ihr das für Eure Zwecke umgestaltet, ist Euch überlassen.

Kommen wir zur Erweiterung der `route.py`. Der neue Code ist das hier:

```
Zeile Inhalt
    1 @main.route("/game/<game_code>")
    2 def game(game_code):
    3     game = Game.query.filter_by(game_code=game_code).first_or_404()
    4     p_actual = session.get("p_actual")
    5     print(f"Spieler: {p_actual}, Spielcode: {game_code}")
    6
    7     # das eigene Spielfeld
    8     my_board  = game.p_1_board if p_actual == game.p_1_name else game.p_2_board
    9     # das gengerische Spielfeld
   10     opp_board = game.p_2_board if p_actual == game.p_1_name else game.p_1_board
   11     p_opponent = ''
   12     if p_actual == game.p_1_name:
   13         p_opponent = game.p_2_name or "noch niemand..."
   14     if p_actual == game.p_2_name:
   15         p_opponent = game.p_1_name
   16
   17     return render_template(
   18         "game.html",
   19         p_1_name=game.p_1_name,
   20         p_2_name=game.p_2_name or "noch niemand...",
   21         game_code=game.game_code,
   22         p_actual=p_actual,
   23         p_opponent=p_opponent,
```

```
24          p1_status=game.p_1_status,
25          p2_status=game.p_2_status,
26          my_board=my_board or "",
27          opp_board=opp_board
28      )
```

Wir holen uns erst das Spiel aus der Instanz unserer Klasse, dazu schauen wir mit dem Spielcode nach.

Aus der `session`, die wir in der `setup`-Route mit den Namen von Spieler 1 oder Spieler 2 belegt haben, lesen wir jetzt den aktuellen Spieler aus, damit können wir auch den Gegner ermitteln.

Dann holen wir uns noch die beiden Spielfelder aus der Instanz der Klasse und geben das zurück an `game.html`.

Jetzt haben wir den ersten Auftritt der `game.js`. Die sieht so aus:

```
Zeile Inhalt
    1 (function () {
    2   'use strict';
    3
    4   /* === Initialisierung ================================================ */
    5   const BOARD_SIZE = 10;
    6
    7   /* === Boards erstellen ============================================== */
    8   function createBoard(root, onClick) {
    9     root.innerHTML = '';
   10     for (let i = 0; i < BOARD_SIZE ** 2; i++) {
   11       const d = document.createElement('div');
   12       d.className = 'cell';
   13       d.dataset.idx = i;
   14       if (onClick) d.addEventListener('click', onClick);
   15       root.appendChild(d);
   16     }
   17   }
   18
   19   /* === Labels für Boards ============================================= */
   20   function generateBoardLabels() {
   21     const rowLabels = document.querySelectorAll('.row-labels');
   22     const colLabels = document.querySelectorAll('.col-labels');
   23
   24     for (const el of rowLabels) {
   25       el.innerHTML = '';
   26       const spacer = document.createElement('div');
   27       spacer.classList.add('label-spacer');
   28       el.appendChild(spacer);
   29       for (let i = 1; i <= 10; i++) {
   30         const div = document.createElement('div');
   31         div.textContent = i;
   32         el.appendChild(div);
   33       }
```

```
34      }
35
36      for (const el of colLabels) {
37        el.innerHTML = '';
38        for (let i = 0; i < 10; i++) {
39          const div = document.createElement('div');
40          div.textContent = String.fromCharCode(65 + i); // A–J
41          el.appendChild(div);
42        }
43      }
44    }
45
46    /* === Setup & Init ====================================================
   */
47    function initGame(userCfg) {
48      const cfg = {
49        ...userCfg,
50        el: {
51          gameCode:   document.getElementById('gameCode'),
52          p_1_name:   document.getElementById('p_1_name'),
53          p_2_name:   document.getElementById('p_2_name'),
54          p_actual:   document.getElementById('p_actual'),
55          p_opponent: document.getElementById('p_opponent'),
56          myBoard:    document.getElementById('myBoard'),
57          oppBoard:   document.getElementById('oppBoard')
58        },
59      };
60
61      console.log("Game initialized with config:", cfg);
62
63      // Erstellen der Boards
64      createBoard(cfg.el.myBoard, null);
65      createBoard(cfg.el.oppBoard, null);
66
67      generateBoardLabels();
68    }
69
70    window.initGame = initGame;
71  })();
```

Auch nicht wenig Code, wo fange ich an? Aus `game.html` wird die `initGame()`-Funktion Zeile 47 gerufen. Ihr werden alle verfügbaren Informationen in der Nutzer-Konfiguration „`userCfg`“ mitgegeben, in `game.html` ist das ganz unten ab Zeile 45 zu finden.

In Zeile 61 geben wir eine Konsolenmeldung aus, die wir bei gestartetem Browser mit der Funktionstaste `F12` ansehen können, das schauen wir uns gleich näher an.

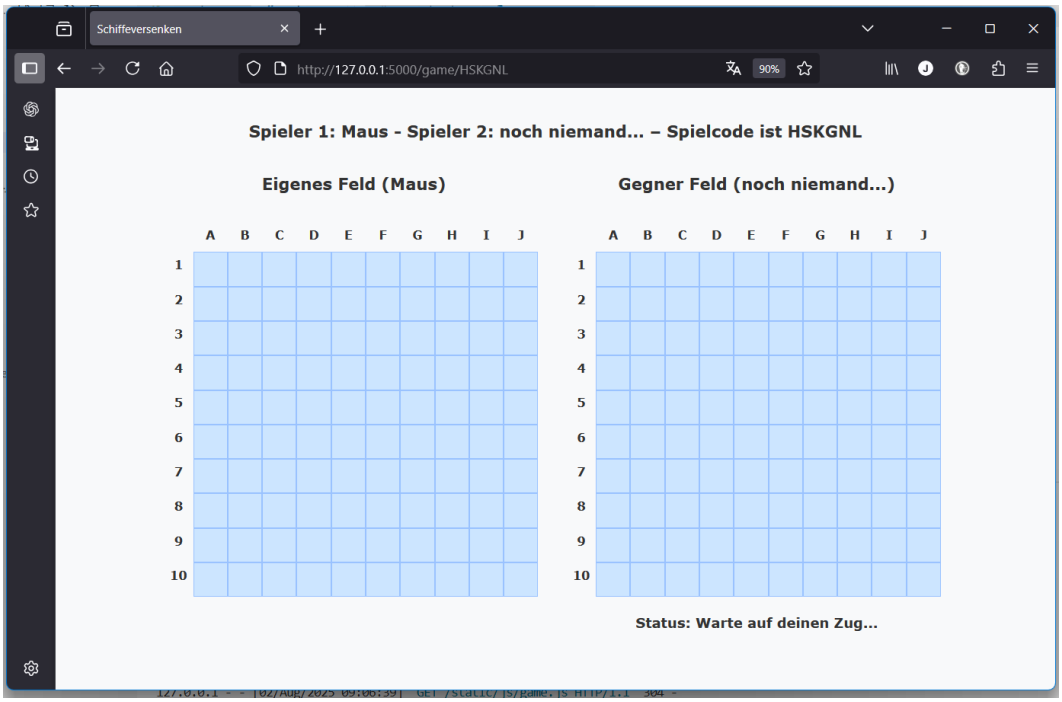
Zeile 64 rufen wir das erste Mal die Funktion `createBoard()` auf und machen das für das eigene Spielfeld, danach das gleiche noch einmal für das gegnerische Feld.

Dann kommen die beiden Rahmen mit Buchstaben und Zahlen mit Hilfe der Funktion `generateBoardLabels()`.

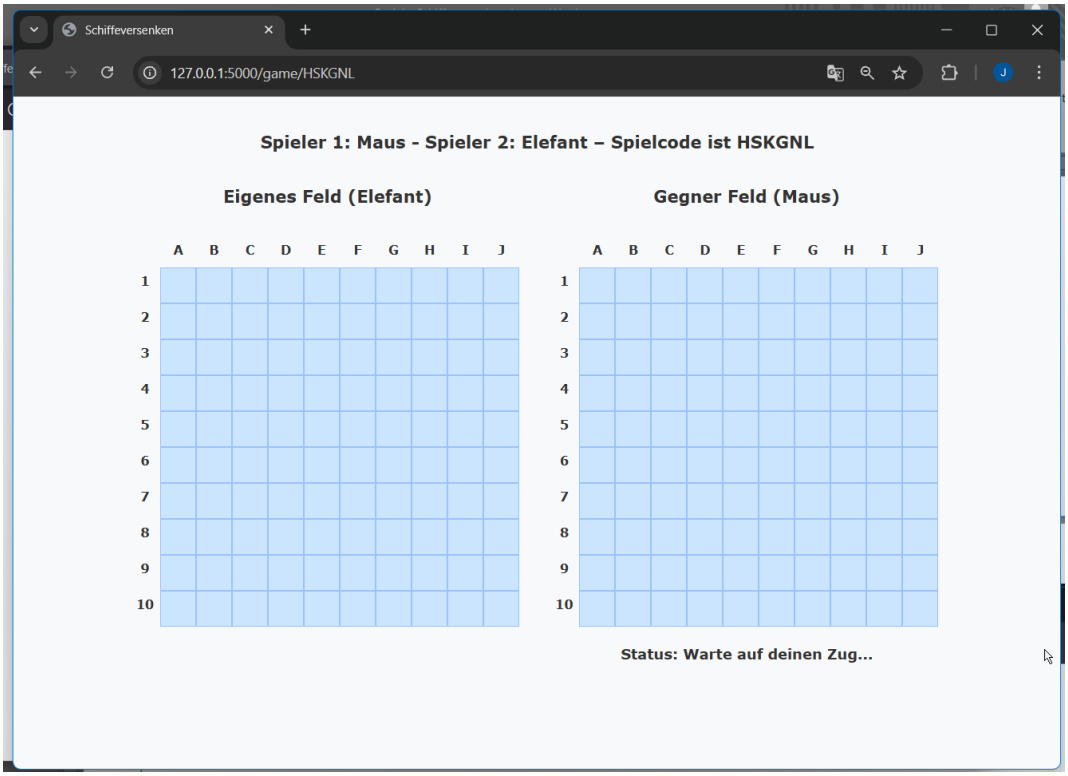
Jetzt kommt der Moment, wo die Kuh den Schwanz hebt, lasst es uns ausprobieren. Spätestens jetzt brauchen wir 2 Browser-Instanzen, damit wir beide Spieler simulieren können.

Spieler 1 ist bei mir die Maus, Spieler 2 der Elefant. Spielcode ist HSKGNL.

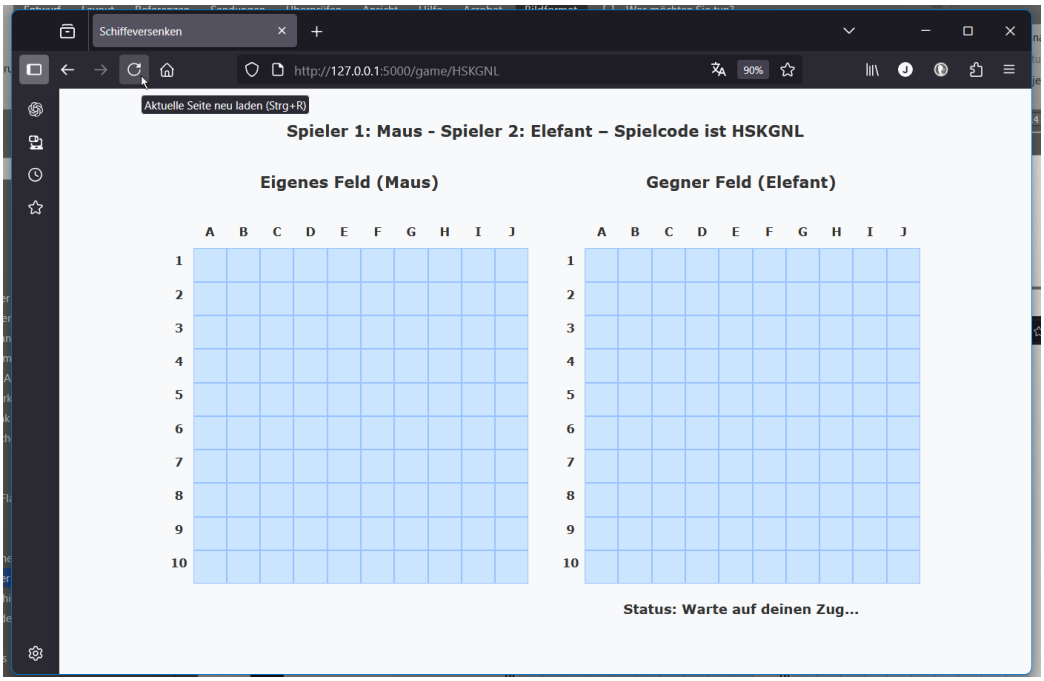
Anmeldung Maus:



Anmeldung Elefant:



Da sich der Elefant nach der Maus angemeldet hat, hat er schon die Information, wer sein Gegner ist, die Maus hatte das zum Zeitpunkt des Screenshots noch nicht. Sobald die Maus den Browser aktualisiert (mit `F5` oder dem Symbol  im Browser), wird ihr der Elefant angezeigt:



Bei Euch sollte das alles genauso aussehen, falls nicht, Ihr wisst ja, `version_04` liegt auf dem Server.

Jetzt müssen wir die ganze Spiellogik einbauen, das wird noch ein schöner Brocken werden, deshalb gibt es jetzt noch einmal eine frische `version_05`.

4.3.6 Setzen der Schiffe

Die Anforderung ist, dass wir 5 Schiffe auf dem eigenen Spielfeld platzieren können müssen. Die Größen der Schiffe sind:

- 1 x 5 Kästchen,
- 1 x 4 Kästchen,
- 2 x 3 Kästchen und
- 1 x 2 Kästchen.

Ich habe mir für das Setzen der Schiffe folgendes überlegt:

- die Reihenfolge ist immer vom größten zum kleinsten Schiff. Zuerst also 5 Kästchen, am Schluss 2 Kästchen.
- Je Schiff kann der Spieler entscheiden, ob er es horizontal oder vertikal setzen möchte. Dafür gibt es 2 Button. Startwert ist „horizontal“.
- Mit Klick auf das Spielfeld wird bei Ausrichtung „horizontal“ das erste Kästchen von links ausgewählt, wenn rechts daneben kein Platz für die restlichen Kästchen des Schiffes ist, kann das Schiff nicht platziert werden. Bei Ausrichtung „vertikal“ wird das erste Kästchen von oben genommen und folgt der gleichen Logik.
- Der Spieler muss die Möglichkeit haben, das Setzen rückgängig zu machen, bis er mit dem erzielten Ergebnis einverstanden ist. Dazu wird ihm ein Button „Zurücksetzen“ angeboten. Dieser macht das jeweils letzte Setzen eines Schiffes rückgängig.

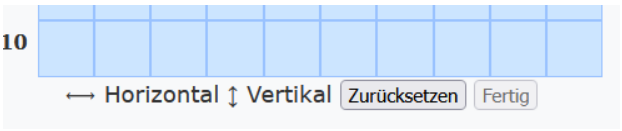
- Sind alle Schiffe platziert, muss der Spieler mit Klick auf den „Fertig“-Button bestätigen, dass er spielbereit ist. Danach ist ein Umsetzen eines Schiffes nicht mehr möglich. Damit der „Fertig“-Button nicht zu früh geklickt werden kann, soll dieser bis zum Platzieren des letzten Schiffs deaktiviert bleiben.
- Das aktuelle Spielfeld wird in der Datenbanktabelle gespeichert.

Viel Arbeit…

Fangen wir mit den 4 Button an. Die erzeugen wir unterhalb des eigenen Spielfelds. Im Code von `game.html` ergänzen wir unten im Abschnitt für das eigene Spielfeld:

```
<div class="game-board">
  <!-- Eigenes Feld -->
  <div class="board-container">
    …
    <div class="button-grid">
      <input type="radio" name="orientation" id="radioHorizontal" value="horizontal"
checked hidden>
      <label for="radioHorizontal">↔ Horizontal</label>
      <input  type="radio"  name="orientation"  id="radioVertical"  value="vertical"
hidden>
      <label for="radioVertical">↕ Vertikal</label>
      <button id="resetBtn">Zurücksetzen</button>
      <button id="readyBtn" disabled>Fertig</button>
    </div>

  </div>

  <!-- Gegnerisches Feld -->
  <div class="board-container">
    …
```

Wenn wir das jetzt ausprobieren, sollten wir die 4 Elemente sehen:



Ihr ahnt es, das müssen wir mit CSS hübsch machen. Nachfolgender Vorschlag zum Aussehen von mir:

```
/* die 4 Button unterhalb des eigenen Spielfelds */
.button-grid {
  display: grid;
  grid-template-columns: repeat(2, 150px);
  grid-gap: 10px;
  justify-content: center;
  margin-top: 20px;
}

.button-grid label,
.button-grid button {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 44px;
  font-size: 14px;
  border: 1px solid #888;
  border-radius: 6px;
  background-color: #007bff;
  color: white;
  cursor: pointer;
```

```
  transition: background-color 0.2s ease;
}

.button-grid label:hover,
.button-grid button:hover {
  background-color: #ccc;
}

.button-grid button:disabled,
.button-grid label.disabled {
  background-color: #ddd;
  opacity: 0.6;
  cursor: not-allowed;
}

/* Verstecke Radio-Inputs */
input[type="radio"][name="orientation"] {
  display: none;
}

/* Aktives Label grün färben */
input[type="radio"][name="orientation"]:checked + label {
  background-color: #c8f7c5; /* Hellgrün */
  border-color: #6cc06c;
  color: #888;
  font-weight: bold;
}

.disabled-label {
  opacity: 0.5;
  pointer-events: none;
  cursor: default;
}
/* Ende der 4 Button unterhalb des eigenen Spielfelds */
```

Zum Testen reicht es, wenn Ihr den Browser noch offen habt, ein Refresh (`F5`) zu machen:

Um der Wahrheit die Ehre zu geben, den Inhalt der CSS-Datei habe ich gemeinsam mit einer KI entworfen, ich verliere unglaublich viel Zeit beim Drehen an den verschiedenen CSS-Stellschrauben. Die KI macht mir einen Vorschlag, ich kopiere das 1 zu 1 in meinen Code und wenn mir das nicht gefällt, sage ich der KI was ich anders haben möchte. Und voilà, das Ergebnis sieht gut aus.

Ich möchte aber auf ein cooles Gadget hinweisen, die beiden Button „Horizontal“ und „Vertikal“ sind eigentlich `Radio`-Button, allerdings sind sie versteckt („`hidden`“ durch „`display: none;`“). Sie haben aber das Aussehen der anderen Button bekommen. Cool, oder? Da hat mir die KI sicher 2 bis 3 Stunden Arbeit abgenommen…

Jetzt kommt der Brocken der neuen `game.js`. Eine Aufteilung in leicht verdauliche Häppchen ist mir nicht geglückt, von daher hier zunächst der gesamte Code:

```
Zeile Inhalt
    1 (function () {
    2   'use strict';
    3
    4   /* === Initialisierung ================================================ */
    5   const BOARD_SIZE = 10;
    6   const SHIPS_TEMPLATE = [
    7     { size: 5, count: 1 },
    8     { size: 4, count: 1 },
    9     { size: 3, count: 2 },
   10     { size: 2, count: 1 },
   11   ];
   12   const TOTAL_SHIPS = SHIPS_TEMPLATE.reduce((s, v) => s + v.count, 0);
   13   const deepCopyShips = () => SHIPS_TEMPLATE.map((s) => ({ ...s }));
   14   const setInfo = (t) => (cfg.el.info.textContent = `${t}`);
   15   const lockBoard = (b) => Array.from(b.children).forEach((c) =>
      c.replaceWith(c.cloneNode(true)));
   16   const unlockBoard = (b, h) => Array.from(b.children).forEach((c) => {
   17     const n = c.cloneNode(true);
   18     n.className = c.className;
   19     n.dataset.idx = c.dataset.idx;
   20     if (h) n.addEventListener('click', h);
   21     c.replaceWith(n);
   22   });
   23   const boardStr = () => boardState.join('');
   24
   25   let cfg = {};
   26   let ships = [];
   27   let boardState = [];
   28   let placedShips = 0;
   29   let placementsStack = [];
   30   let orientation = 'horizontal';
   31
   32   /* === Hilfsfunktion für JSON ======================================== */
   33   async function postJSON(url, body) {
   34     const res = await fetch(url, {
   35       method: 'POST',
   36       headers: { 'Content-Type': 'application/json' },
   37       body: JSON.stringify(body),
   38     });
   39     if (!res.ok) throw new Error(`HTTP ${res.info}`);
   40     return res.json();
   41   }
   42
   43   /* === Boards erstellen ============================================== */
   44   function createBoard(root, onClick) {
   45     root.innerHTML = '';
   46     for (let i = 0; i < BOARD_SIZE ** 2; i++) {
   47       const d = document.createElement('div');
```

```
      d.className = 'cell';
      d.dataset.idx = i;
      if (onClick) d.addEventListener('click', onClick);
      root.appendChild(d);
    }
  }

  /* === Labels für Boards ============================================= */
  function generateBoardLabels() {
    const rowLabels = document.querySelectorAll('.row-labels');
    const colLabels = document.querySelectorAll('.col-labels');

    for (const el of rowLabels) {
      el.innerHTML = '';
      const spacer = document.createElement('div');
      spacer.classList.add('label-spacer');
      el.appendChild(spacer);
      for (let i = 1; i <= 10; i++) {
        const div = document.createElement('div');
        div.textContent = i;
        el.appendChild(div);
      }
    }

    for (const el of colLabels) {
      el.innerHTML = '';
      for (let i = 0; i < 10; i++) {
        const div = document.createElement('div');
        div.textContent = String.fromCharCode(65 + i); // A–J
        el.appendChild(div);
      }
    }
  }

  /* === Platzierung ================================================== */
  const nextShipSize = () => ships.find((s) => s.count > 0)?.size ?? null;
  function canPlace(start, size) {
    const r0 = Math.floor(start / BOARD_SIZE);
    const c0 = start % BOARD_SIZE;

    for (let i = 0; i < size; i++) {
      const r = orientation === 'horizontal' ? r0 : r0 + i;
      const c = orientation === 'horizontal' ? c0 + i : c0;

      // Grenzen prüfen
      if (r >= BOARD_SIZE || c >= BOARD_SIZE) return false;

      // Index der aktuellen Zelle
      const currentIndex = r * BOARD_SIZE + c;

      // Direkt belegte Felder verhindern
```

```
      if (cfg.el.myBoard.children[currentIndex].classList.contains('ship')) return false;

      // Nur orthogonale Nachbarn (oben, unten, links, rechts) prüfen
      const orthogonalNeighbors = [
        [r - 1, c], // oben
        [r + 1, c], // unten
        [r, c - 1], // links
        [r, c + 1]  // rechts
      ];

      for (const [nr, nc] of orthogonalNeighbors) {
        if (nr < 0 || nc < 0 || nr >= BOARD_SIZE || nc >= BOARD_SIZE) continue;
        const neighborIndex = nr * BOARD_SIZE + nc;
        if (cfg.el.myBoard.children[neighborIndex].classList.contains('ship')) {
          setInfo('Schiff dürfen sich nicht berühren!');
          return false;
        }
      }
    }
    setInfo('Schiff platziert');
    return true;
  }
  const paintShip = (start, size) => {
    for (let i = 0; i < size; i++) {
      const off = orientation === 'horizontal' ? i : i * BOARD_SIZE;
      cfg.el.myBoard.children[start + off].classList.add('ship');
      boardState[start + off] = 1;
    }
  };

  function placeShip(e) {
    if (placedShips >= TOTAL_SHIPS) return;
    const idx = +e.target.dataset.idx;
    const size = nextShipSize();
    if (!size || !canPlace(idx, size)) return;
    paintShip(idx, size);
    ships.find((s) => s.size === size).count--;
    placedShips++;

    placementsStack.push({
      index: idx,
      size: size,
      orientation: orientation
    });
    if (placedShips === TOTAL_SHIPS) {
      lockBoard(cfg.el.myBoard);
      cfg.el.readyBtn.disabled = false;
      setInfo('Alle Schiffe platziert - Fertig!');
    }
  }
  function removeShip(start, size, orientation) {
```

```
150     for (let i = 0; i < size; i++) {
151       const off = orientation === 'horizontal' ? i : i * BOARD_SIZE;
152       cfg.el.myBoard.children[start + off].classList.remove('ship');
153       boardState[start + off] = 0;
154     }
155
156     const ship = ships.find(s => s.size === size);
157     if (ship) {
158       ship.count++; // Zähler erhöhen
159     }
160
161     placedShips--; // Verringere die Anzahl der platzierten Schiffe
162
163     console.log("Schiff entfernt:", { start, size, orientation, placedShips });
164
165     if (placedShips < TOTAL_SHIPS) {
166       cfg.el.readyBtn.disabled = true;
167       unlockBoard(cfg.el.myBoard, placeShip);
168     }
169   }
170
171   /* === Buttons ============================================================ */
172   function initButtons() {
173     cfg.el.orientationRadios.forEach(radio => {
174       radio.addEventListener('change', (e) => {
175         orientation = e.target.value;
176       });
177     });
178
179     cfg.el.resetBtn.addEventListener('click', () => {
180       if (placementsStack.length > 0) {
181         // Entferne die letzte Platzierung
182         const lastPlacement = placementsStack.pop();
183         // Entferne das Schiff von der Anzeige
184         removeShip(lastPlacement.index, lastPlacement.size,
    lastPlacement.orientation);
185         setInfo('Letzte Platzierung rückgängig gemacht.');
186       } else {
187         setInfo('Keine Platzierungen zum Rückgängigmachen.');
188       }
189
190       // Radio-Buttons reaktivieren und „horizontal" als Standard setzen
191       cfg.el.orientationRadios.forEach(radio => {
192         radio.disabled = false;
193         radio.checked = radio.value === 'horizontal';
194       });
195       document.getElementById('radioHorizontal').checked = true;
196       orientation = 'horizontal';
197     });
198
199     cfg.el.readyBtn.addEventListener('click', () => {
200       cfg.el.readyBtn.disabled = true;
```

```
201       cfg.el.resetBtn.disabled = true;
202
203       // Alle Radio-Buttons deaktivieren
204       cfg.el.orientationRadios.forEach(radio => {
205         radio.disabled = true;
206
207         // Zugehöriges Label visuell deaktivieren
208         const label = document.querySelector(`label[for="${radio.id}"]`);
209         if (label) {
210           label.classList.add('disabled-label');
211         }
212       });
213
214       lockBoard(cfg.el.myBoard);
215
216       postJSON('/save_board', {
217         gameCode: cfg.gameCode,
218         myBoard: boardStr(),
219         p_actual: cfg.player
220       })
221         .then(() => setInfo('Spielbrett gesendet.'))
222         .catch(console.error);
223     });
224
225   }
226   /* === Setup & Init ====================================================== */
227   function startPlacement() {
228     ships = deepCopyShips();
229     boardState = Array(BOARD_SIZE ** 2).fill(0);
230     placedShips = 0;
231     placementsStack = [];
232     orientation = 'horizontal';
233     createBoard(cfg.el.myBoard, placeShip);
234     cfg.el.readyBtn.disabled = true;
235   }
236   function initGame(userCfg) {
237     cfg = {
238       ...userCfg,
239       el: {
240         gameCode:   document.getElementById('gameCode'),
241         p_1_name:   document.getElementById('p_1_name'),
242         p_2_name:   document.getElementById('p_2_name'),
243         p_actual:   document.getElementById('p_actual'),
244         p_opponent: document.getElementById('p_opponent'),
245         myBoard:    document.getElementById('myBoard'),
246         oppBoard:   document.getElementById('oppBoard'),
247         resetBtn:   document.getElementById('resetBtn'),
248         readyBtn:   document.getElementById('readyBtn'),
249         info:       document.getElementById('info'),
250         orientation: 'horizontal',
251         //orientationRadios:
    Array.from(document.querySelectorAll('input[name="orientation"]')),
```

```
252         orientationRadios: document.querySelectorAll('input[name="orientation"]'),
253       },
254     };
255
256     console.log("Game initialized with config:", cfg);
257
258     // Erstellen der Boards
259     createBoard(cfg.el.oppBoard, null);
260
261     generateBoardLabels();
262     initButtons();
263     startPlacement();
264   }
265
266   window.initGame = initGame;
267 })();
268   }
269
270   // Globale Initialisierungsfunktion
271   window.initGame = initGame;
272 })();
```

Ich sagte ja, dass das ein Brocken ist...

Gehen wir es durch. Mein Tipp, legt Euch das nebeneinander auf den Bildschirm, auf der einen Seite den Code, auf der anderen Seite diese Doku.

Ab Zeile 5 geht es mit der Deklaration der globalen Konstanten („`const`“) und Variablen („`let`“) los. Zuerst legen wir die Größe und die jeweilige Anzahl der Schiffe fest. `deepCopyShips` erstellt eine Kopie der Schiffe, um das Original zu schützen. `setInfo()` setzt den anzuzeigenden Text unterhalb des gegnerischen Spielfelds. `lockBoard()` und `unlockBoard()` werden dafür genutzt, dass nach Klick auf den Fertig-Button oder nach einem Wassertreffer das jeweilige Board für Eingaben gesperrt oder geöffnet wird.

Die Variablen werden leer zur Verfügung gestellt.

Die `async`-Funktion wird als Standard für die Kommunikation zwischen der HTML-Seite und den Routes genutzt.

Die nächsten beiden Funktionen sind schon bekannt, neu ist die Platzierung der Schiffe ab Zeile 83. `nextShipSize()` und `paintShip()` (Zeile 121) werden in `placeShip()` (Zeile 129) benötigt. Zusammen sorgen sie dafür, dass

- die Liste der Schiffe abgearbeitet wird,
- die jeweilige Größe gelesen wird,
- das zu platzierende Schiff
    - in der angegebenen Richtung auf das Spielfeld passt
    - sich nicht mit einem anderen Schiff kreuzt oder
    - direkt an ein anderes Schiff anschließt
- und wenn alle Schiffe platziert wurden,
    - der Fertig-Button aktiviert wird und
    - eine entsprechende Meldung ausgegeben wird

Mit `removeShip()` kann bei Klick auf den Zurücksetzen-Button das jeweils letzte Platzieren rückgängig gemacht werden.

Die ganzen Funktionen werden in `startPlacement()` aufgerufen (Zeile 227), die wiederum in der `initGame()` in Zeile 263 gerufen wird.

Das Platzieren der Schiffe selbst beginnt in Zeile 129 mit `placeShip()`. Die Deklarationen und Funktionen ab Zeile 118 werden von `placeShip()` gerufen. Sobald alle Schiffe platziert sind, wird erst der Fertig-Button anklickbar (Zeile 145).

Die Funktion `initButton()` ab Zeile 172 klammert die Aktivitäten rund um die beiden Button „Zurücksetzen“ und „Fertig“.

Mit dem Zurücksetzen wird das letzte platzierte Schiff aus der Liste gelöscht. Sollte der „Fertig“-Button schon klickbar sein (also *nach* Setzen des 5. Schiffes aber *vor* Klicken auf den „Fertig“-Button) wird auch der „Fertig“-Button wieder gesperrt.

Sobald der „Fertig“-Button geklickt wird

- werden alle Button auf inaktiv gesetzt
- das eigene Spielfeld wird für weitere Eingaben geperrt
- übergibt das eigene Spielfeld an die Route `/save_board` und
- gibt die Meldung aus, dass das Spielfeld gesendet wurde

Damit hätten wir die `game.js` besprochen, wir brauchen jetzt noch die Änderungen in der `route.py`.

Für einen schnellen Test ohne Speicherung in der Datenbank machen wir uns eine rudimentäre route `/save_board`. Die sieht so aus:

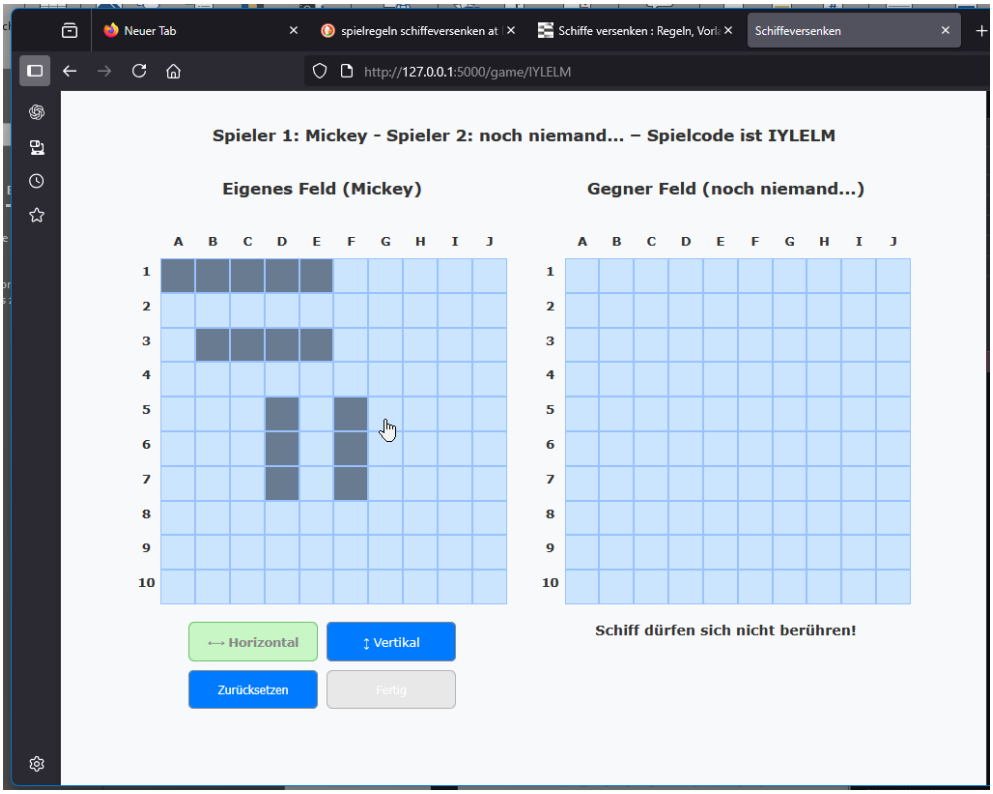
Zeile	Inhalt
1	`@main.route("/save_board", methods=["POST"])`
2	`def save_board():`
3	`    data = request.get_json(force=True)`
4	`    game_code  = data.get("game_code")`
5	`    my_board   = data.get("my_board")`
6	`    p_actual   = data.get("p_actual")`
7	`    db_action  = data.get("db_action")`
8	
9	`    if not (gameCode and board and p_actual) or len(board) != 100:`
10	`        return jsonify({"error": "Ungültige Daten"}), 400`
11	
12	`    return jsonify({"status": "ok"}), 200`

Wir brauchen auch noch den `import` von `jsonify` in der `route.py`, also ganz oben ergänzen:

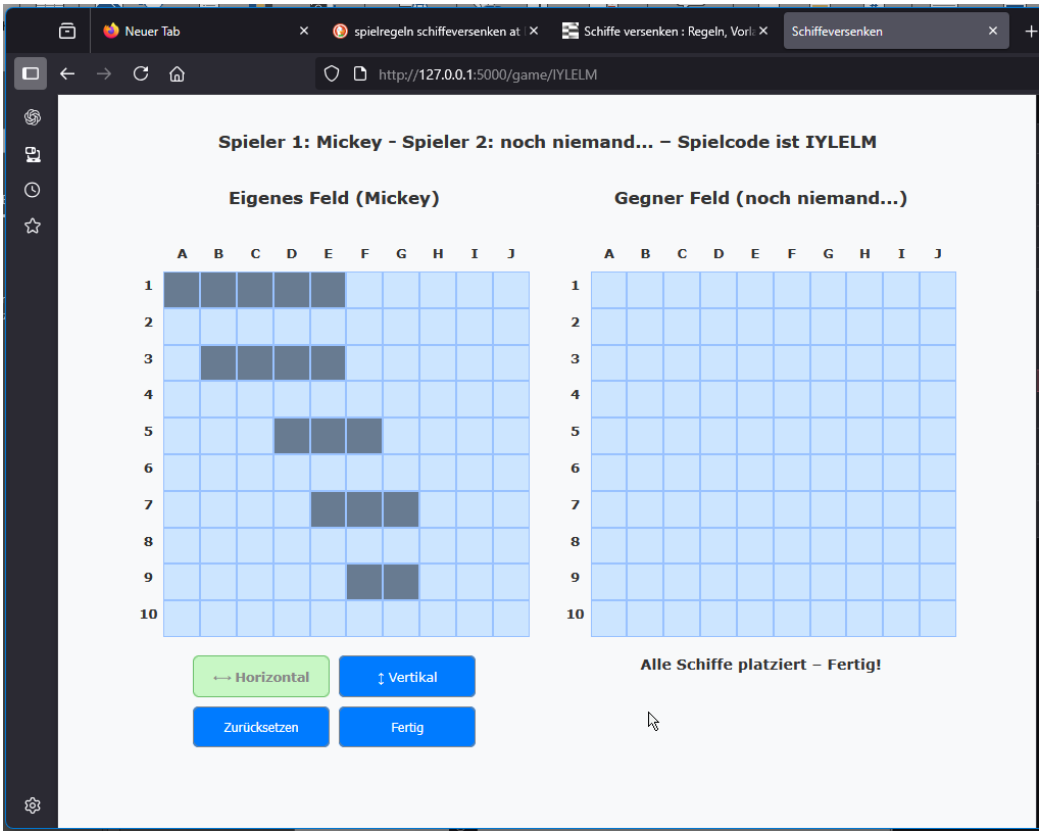
```
from flask import Blueprint, render_template, request, redirect, url_for, session, jsonify
```

Das war es dann auch schon, zumindest für den Moment, wenn wir nachher das Spiel richtig starten, kommt da noch was dazu, das aber später…

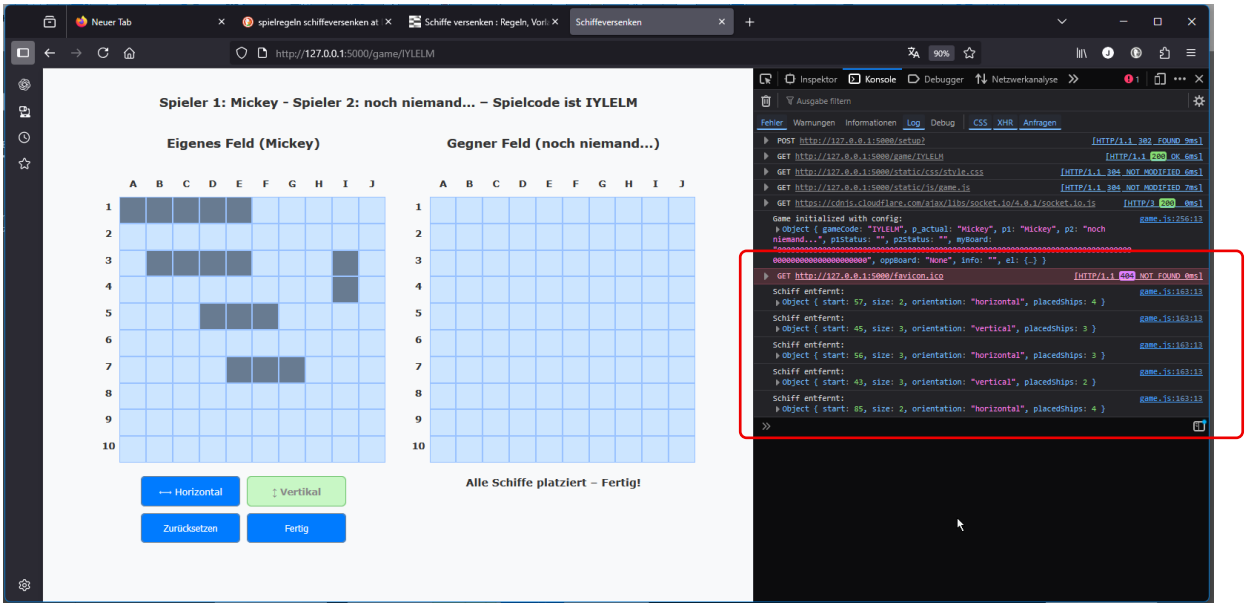
Dann wollen wir mal sehen, was wir da fabriziert haben. Nach Neustart des Servers melden wir uns als Spieler 1 an:



Alle 5 Schiffe platziert und der Fertig-Button ist aktiv:



Das Rückgängigmachen klappt, auch mehrfach:



In der Datenbank ist natürlich noch nichts passiert, das gehen wir jetzt an. Spendieren wir uns eine neue `version_06`, damit wir zur Not auf den alten Stand zurückrollen können.

4.3.7 Das Spielen

Mit Klick auf den „Fertig“-Button startet das eigentliche Spiel, der Spieler signalisiert damit, dass er alle seine Schiffe nach seinen Wünschen platziert hat und jetzt spielbereit ist. Dabei ist es unerheblich, ob er Spieler 1 oder Spieler 2 ist und wie weit sein Gegner in den Vorbereitungen ist.

Der Spielablauf wird folgender sein:

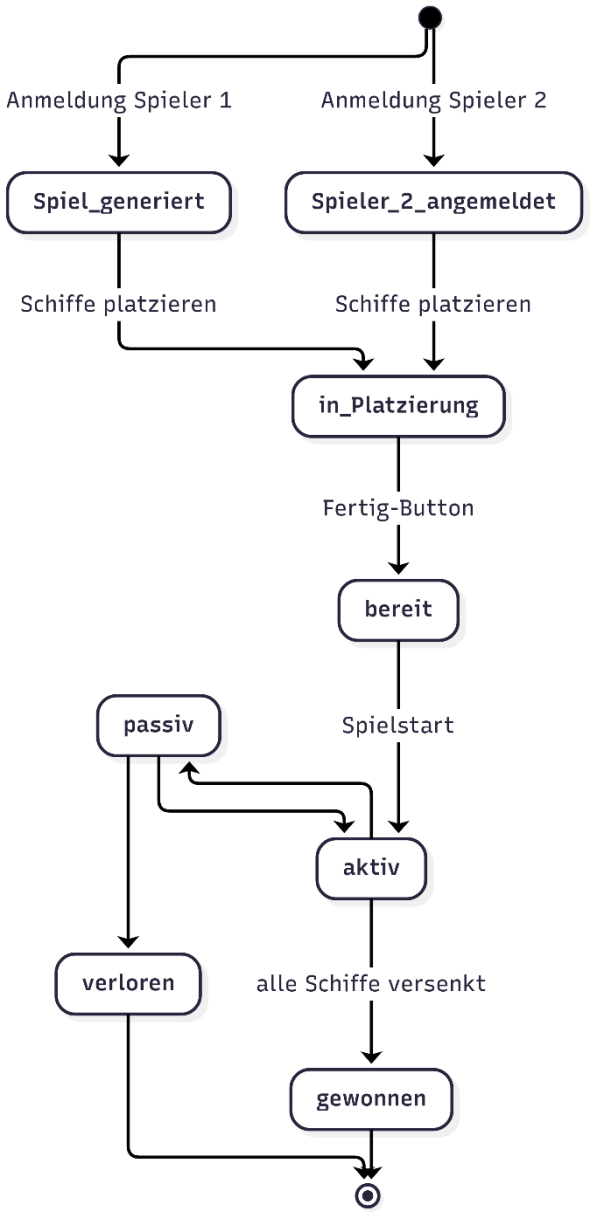
Zuerst sorgen wir dafür, dass:

- das neue Spielfeld in der Datenbanktabelle für den jeweiligen Spieler eingetragen wird,
- der Status des Spielers aktualisiert wird, zu diesem Zeitpunkt ist das „bereit“

Ist bei beiden Spielern der Status „bereit“,

- setzen wir per Zufall bei einem Spieler den Status „aktiv“ bei dem anderen „passiv“.
- Ist bei dem aktuellen Spieler der Status „aktiv“,
    - öffnen wir das gegnerische Spielfeld für den Schuss,
    - prüfen, ob der Schuss ein „Treffer“ oder „Wasser“ ist,
    - aktualisieren das generische Spielfeld entsprechend mit einem „T“ oder einem „W“,
    - bei „Wasser“ wechseln wir den Status von „aktiv“ auf „passiv“ und
    - speichern den neuen Stand in der Datenbanktabelle

Um den Statusübergang zu visualisieren, hier ein Bildchen dazu:



Einstieg ist die Anmeldung entweder als Spieler 1 oder Spieler 2.

Ist der aktuelle Spieler die Nummer 1, wird ein neues Spiel für ihn generiert der Status ist dann „Spiel generiert“. Spieler 2 tritt dem aktuellen Spiel bei, sein Status ist dann „Spieler 2 angemeldet“.

Haben sich 2 Spieler in einem Spiel angemeldet, wechselt der Status für beide Spieler auf „in Platzierung“. Solange dieser Status besteht, kann jedes Schiff beliebig oft gelöscht und wieder gesetzt werden. Sind alle Schiffe nach Wunsch platziert, muss der „Fertig“-Button geklickt werden.

Sobald der Spieler den „Fertig“-Button gedrückt hat, wechselt sein Status auf „bereit“.

Sind beide Status „bereit“, erhält einer der Spieler per Zufall den Status „aktiv“, der andere den Status „passiv“.

Nur im Status „aktiv“ kann ein Schuss abgegeben werden, dieser bleibt bestehen, solange der Schuss ein „Treffer“ ist. Bei „Wasser“ wechseln die Status der beiden Spieler.

Ist das letzte Schiff vollständig getroffen, wechselt der Status des aktiven Spielers auf „gewonnen“ und des passiven Spielers auf „verloren“, das Spiel ist zu Ende.

Soweit klar, oder?

Jetzt müssen wir uns noch ein Gedanken zum Ablauf des Spiels machen. Wie realisierten wir das Schießen? Wie erfährt der Gegner vom Schuss auf sein Spielfeld? Wie erfolgt der Spielerwechsel?

Die Speicherung in der Datenbanktabelle wird in der `routes.py` initiiert, die Kommunikation mit der HTML-Seite übernimmt das JavaScript `game.js`. Das sind also die beiden Komponenten, die Informationen austauschen und entsprechende Handlungen durchführen.

Dies zu dokumentieren wird schwierig, ich werde den passenden Code-Schnipsel mal unterhalb des Spiegelstriches unterbringen.

Was passiert also wo? Schauen wir uns das an.

- Der Fertig-Button wird in der `game.js` abgefangen und muss an die `route.py` zur Speicherung weitergegeben werden an `/save_board`. Ausschnitt aus der `game.js`:

```
  function initButtons() {
…
    cfg.el.readyBtn.addEventListener('click', () => {
      cfg.el.readyBtn.disabled = true;
      cfg.el.resetBtn.disabled = true;
…
      postJSON('/save_board', {
        game_code: cfg.game_code,
        my_board: boardStr(),
        p_actual: cfg.p_actual,
        db_action: 'Platzierung fertig'
      })
```

- In der `route.py` in `/save_board` wird der Status „bereit“ gesetzt. Sind beide Spieler auf „bereit“, wird einem Spieler per Zufall der Status „aktiv“ vergeben, dem anderen „passiv“, dann geht es zurück zur `game.html` und damit zur `game.js`.

```
@main.route("/save_board", methods=["POST"])
def save_board():
…
    if db_action == "in Platzierung":
        # Spieler ist in der Platzierungsphase
        if p_actual == game.p_1_name:
            game.p_1_board = my_board
            game.p_1_status = "in Platzierung"
        else:
            game.p_2_board = my_board
            game.p_2_status = "in Platzierung"
    elif db_action == "Platzierung fertig":
        # Spieler hat die Platzierung abgeschlossen
        if p_actual == game.p_1_name:
            game.p_1_board = my_board
            game.p_1_status = "bereit"
        else:
            game.p_2_board = my_board
            game.p_2_status = "bereit"

    # Startspieler setzen, wenn beide Spieler bereit sind
    if game.p_1_status == "bereit" and game.p_2_status == "bereit":
        startspieler = random.choice([game.p_1_name, game.p_2_name])
        #print(f"Startspieler ist: {startspieler}")
        if startspieler == game.p_1_name:
            game.p_1_status = "aktiv"
            game.p_2_status = "passiv"
```

```
        else:
            game.p_1_status = "passiv"
            game.p_2_status = "aktiv"
    lockBoard(cfg.el.my_board);
```

- In `game.js` müssen wir nun eine neue Funktion implementierten („`shoot()`“), die das Schießen übernimmt. Ist der Status des aktuellen Spielers „aktiv“, muss
  - das Spielfeld des Gegners geöffnet werden und
  - der Spieler die Möglichkeit haben, dort ein Feld anzuklicken.

```
function shoot(e) {
  const idx = +e.target.dataset.idx;
  const cell0 = cfg.el.opp_board.children[idx];
  if (cell0.classList.contains('hit') || cell0.classList.contains('miss')) return;
…
function shoot(e) {
  const idx = +e.target.dataset.idx;
  const cell0 = cfg.el.opp_board.children[idx];
  if (cell0.classList.contains('hit') || cell0.classList.contains('miss')) return;

  lockBoard(cfg.el.opp_board);
  postJSON('/shoot', { game_code: cfg.game_code, index: idx, p_actual: cfg.p_actual })
    .then((d) => {
      const cell = cfg.el.opp_board.children[idx];
      if (d.result === 'T') {
        cell.classList.add('hit');
        setInfo('Treffer!');
      } else {
        cell.classList.add('miss');
        setInfo(`Wasser – ${d.next_player || 'Gegner'} ist dran`);
      }
      if (d.result === 'T' && !d.game_over) unlockBoard(cfg.el.opp_board, shoot);
…
```

- Hat der Spieler das getan, muss der Schuss in der Datenbank festgehalten werden, dafür machen wir uns in `routes.py` eine neue Route `/shoot`. In `/shoot`
  - bekommen wir die Position des Schusses,
  - lesen zum aktuellen Spieler das gegnerische Spielfeld und
  - prüfen, ob dort ein Teil eines Schiffes liegt.
  - Wenn nein,
    - wird der Schuss mit „W“ für Wasser in der Datenbanktabelle gespeichert (also die existierende „0“ wird mit „W“ überschrieben) und
    - der Status der beiden Spieler gewechselt, „aktiv“ wird zu „passiv“ und anders herum
  - wenn ja,
    - wird der Schuss mit „T“ für Treffer in der Datenbanktabelle gespeichert (also die existierende „1“ wird mit „T“ überschrieben), die beiden Status bleiben unverändert, der aktuelle Spieler darf noch einmal schießen.

- Dann geben wir die Steuerung wieder an `game.js` ab.
- Die Aufbereitung der beiden Spielfelder erfolgt dann in der `game.html`

```
@main.route("/shoot", methods=["POST"])
def shoot():
    data      = request.get_json(force=True)
    game_code = data.get("game_code")
    idx       = data.get("index")
    p_actual  = data.get("p_actual")

    # 1) Validierung
    if game_code is None or idx is None or p_actual is None:
        return jsonify({"error": "Fehlende Daten"}), 400
    if not (0 <= idx < 100):
        return jsonify({"error": "Index außerhalb des Spielfelds"}), 400

    # 2) Spiel laden
    game = Game.query.filter_by(game_code=game_code).first()
    if not game:
        return jsonify({"error": "Spiel nicht gefunden"}), 404

    # 3) Wer ist dran, welches Board?
    if p_actual == game.p_1_name and game.p_1_status == "aktiv":
        target_board = list(game.p_2_board or "0" * 100)
        shooter, target, next_status = "p_1_status", "p_2_status", ("passiv",
"aktiv")
    elif p_actual == game.p_2_name and game.p_2_status == "aktiv":
        target_board = list(game.p_1_board or "0" * 100)
        shooter, target, next_status = "p_2_status", "p_1_status", ("passiv",
"aktiv")
    else:
        return jsonify({"error": "Nicht dein Zug"}), 403

    # 4) Treffer?
    hit = target_board[idx] == "1"
    result = "T" if hit else "W"
    target_board[idx] = "T" if hit else "W"

    # Board wieder sichern
    if shooter == "p_1_status":
        game.p_2_board = "".join(target_board)
        db.session.commit()
    else:
        game.p_1_board = "".join(target_board)
        db.session.commit()

    # 5) Spieler wechseln
    if not hit:
        setattr(game, shooter, "passiv")
```

```
            setattr(game, target,  "aktiv")
        db.session.commit()
         …
        next_player = game.p_2_name if shooter == "p_1_status" else game.p_1_name
        return jsonify({"result": result, "next_player": next_player})
```

- Das Lesen der Datenbanktabelle geben wir wieder an die `route.py`, dafür bauen wir uns in der `game.js` eine neue Funktion „`poll()`“, die dafür sorgt, dass das Lesen regelmäßig alle 3 Sekunden passiert:

```
  function poll() {
    postJSON('/get_status', { game_code: cfg.game_code, p_actual: cfg.p_actual })
      .then((d) => {
…
        } else {
          disableOpp();
          setInfo('Warte auf deinen Zug …');
          const fast = d.my_status === 'passiv';
          schedulePoll(fast ? 1000 : 3000);
…
```

- In der `route.py` dort die neue Route `/get_status`. Wir übergeben den Spielcode und den aktuellen Spieler und erhalten passgenau das eigene Spielfeld und das des Gegners zurück:

```
@main.route("/get_status", methods=["POST"])
def get_status():
    data      = request.get_json(force=True) or {}
    game_code = data.get("game_code")
    p_actual  = data.get("p_actual")

    # Grundprüfung
    if not game_code or not p_actual:
        return jsonify({"error": "Fehlende Daten"}), 400

    # Spiel laden
    game = Game.query.filter_by(game_code=game_code).first()
    if not game:
        return jsonify({"error": "Spiel nicht gefunden"}), 404

    # ermitteln Gegner
    if p_actual == game.p_1_name:
        p_1_name = game.p_1_name
        p_2_name = game.p_2_name
        my_status   = game.p_1_status
        my_board = game.p_1_board
        opp_board = mask_board(game.p_2_board)
        #reload_flag = True
    else:
        p_1_name = game.p_1_name
```

```
            p_2_name = game.p_2_name
            my_status   = game.p_2_status
            my_board = game.p_2_board
            opp_board = mask_board(game.p_1_board)
    …
```

- Zurück in der `game.js` bereiten wir die beiden Spielfelder auf.
    - Auf dem *eigenen* Spielfeld des passiven Spielers wird der Schuss angezeigt, bei Treffer rot, bei Wasser blau:

```
if (d.my_board) markOpponentHits(d.my_board);
```

    - Auf dem *gegnerischen* Spielfeld des aktiven Spielers wird ebenfalls ein Treffer rot und ein Wasserschuss mit blau markiert:

```
if (d.opp_board) markMyShots(d.opp_board);
```

    - Der Hinweistext unter dem Spielfeld des Gegners wird aktualisiert und der nächste Spieler wird ermittelt. Wenn der Schuss ins Wasser ging, ist der Gegner dran:

```
if (d.result === 'T') {
  cell.classList.add('hit');
  setInfo('Treffer!');
} else {
  cell.classList.add('miss');
  setInfo(`Wasser – ${d.next_player || 'Gegner'} ist dran`);
}
```

- In der `shoot`-Route wird zuletzt geprüft, wer gewonnen hat.
    - Dazu wird immer geprüft, ob im gegnerischen Spielfeld noch eine „1“ enthalten ist
    - Gibt es dort keine mehr, hat der aktive Spieler alle Schiffe des Gegners getroffen und somit gewonnen. Der Status des aktuellen Spielers wird auf „gewonnen“ geändert, der des Gegners auf „verloren“.
    - Dann wird eine neue Route `/gameOver` aufgerufen.
    - Dieser wird der Spielcode und der aktuelle Spieler mitgegeben.

```
 if "1" not in target_board:
       game.p_1_status = "gewonnen" if shooter == "p_1_status" else "verloren"
       game.p_2_status = "verloren" if shooter == "p_1_status" else "gewonnen"
       db.session.commit()
       return redirect(url_for("main.gameOver", game_code=game_code,
       p_actual=p_actual))
```

- In der `/gameOver`-Route wird nun über den Spielcode
    - das Spiel,
    - der aktuelle Spieler,
    - der Gegner,
    - der eigene Status,
    - das eigene Spielfeld und
    - das Spielfeld des Gegners gelesen.
    - Diese Informationen werden an das HTML-Dokument `gameOver.html` übergeben

```
    return render_template(
        "gameOver.html",
        outcome=outcome,
        p_actual=p_actual,
        game_code=game_code,
        my_board=my_board,
        opp_board=opp_board,
        opponent=opponent
    )
```

- In `gameOver.html` findet die Aufbereitung des Ergebnisses statt.
    - Das Ergebnis ist unterschiedlich, je nachdem ob der Spieler gewonnen oder verloren hat,
    - es werden aber beide Spielfelder mit den Positionen der Schiffe und den bisher abgegebenen Schüssen angezeigt, sodass jeder Spieler sehen kann, wo die Schiffe des Gegners versteckt waren.
    - Zu guter Letzt wird beiden Spielern noch ein Button angeboten, über den man ein neues Spiel starten kann.

Damit sind wir, was das Coden anbelangt am Ende.

Den fertigen Code werde ich hier nicht kopieren, das würde das Dokument nur unnötig aufblähen. Bitte ladet Euch das Ergebnis der `version_06` aus dem Ordner mit dem Quellcode.

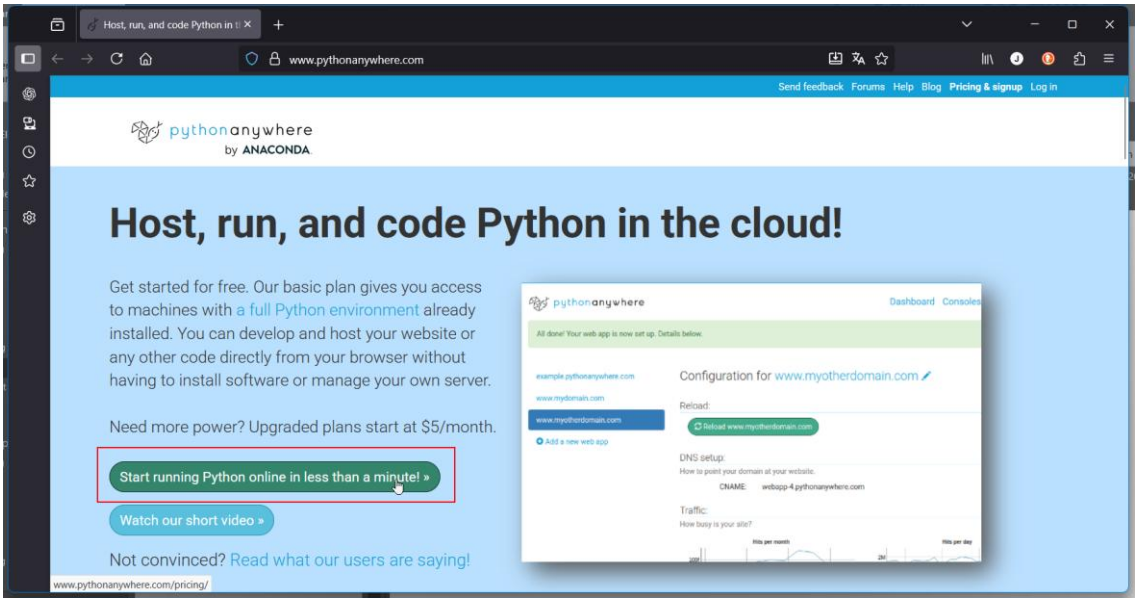
Was ich aber noch zeigen will ist, wie Ihr Eure Variante im Internet hosten könnt. Das wird dann das Ende dieses Projekts sein.

4.4 Die Veröffentlichung

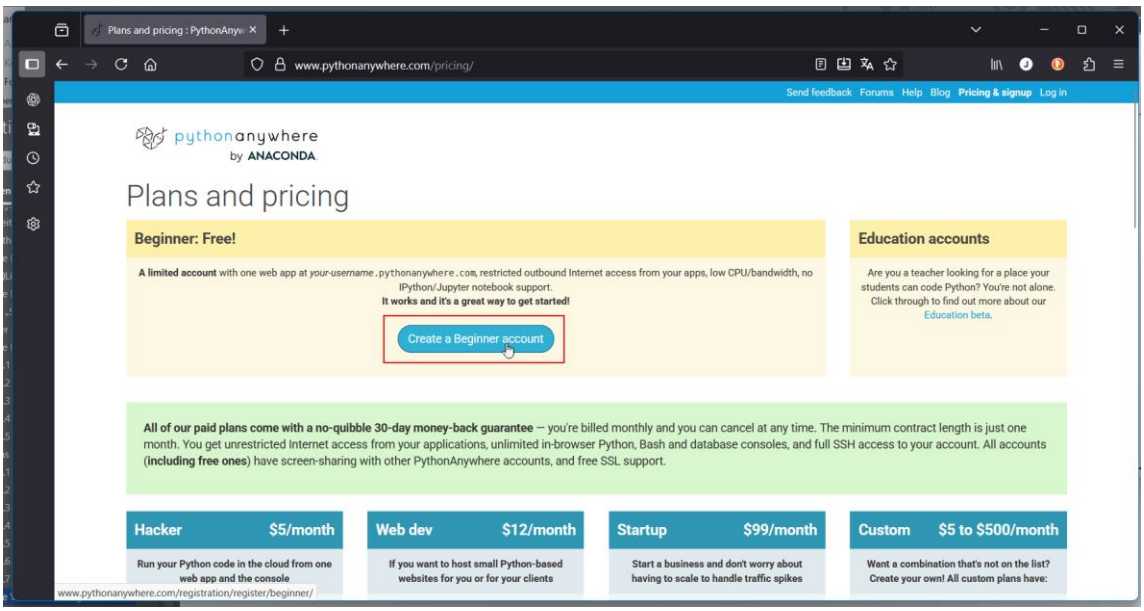
Wie versprochen, hier noch die Anleitung, wie Ihr das Spiel Online bringen könnt. Ich habe mich für Pythonanywhere entschieden, es gibt aber auch noch andere Anbieter.

Wir bringen also die aktuelle `version_06` unter https://papasbattleship.pythonanywhere.com/ ins Internet. Pythonanywhere selbst ist über den Link http://pythonanywhere.com zu erreichen.

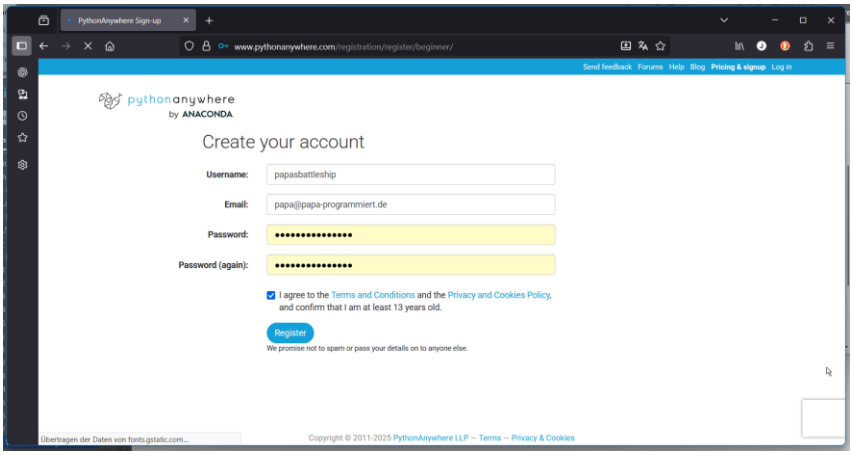
Im Startbildschirm auf „Start running…“ klicken:

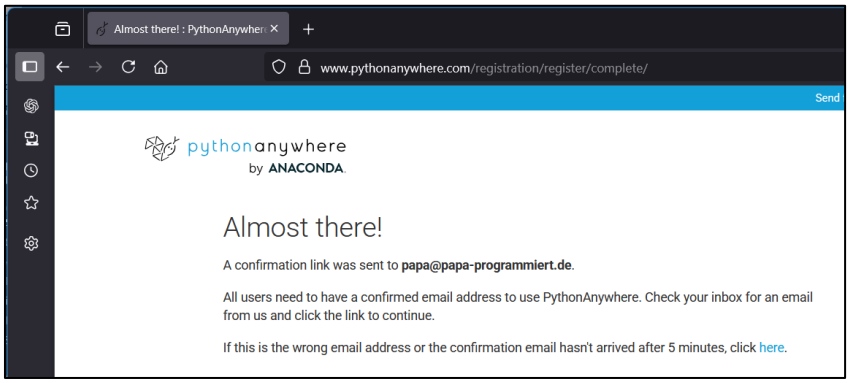


Dann „Create a Beginner account“:

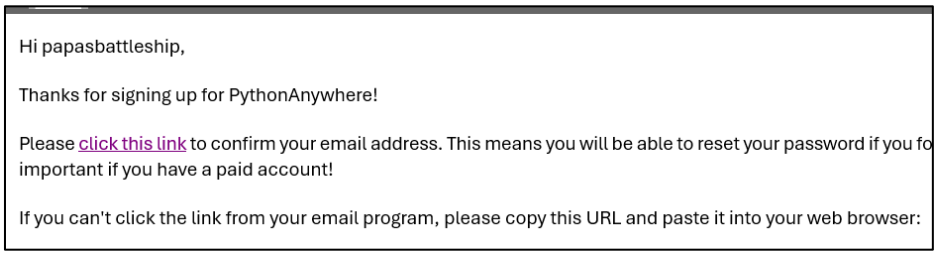


Die eigene Mail-Adresse und ein starkes Passwort auswählen:

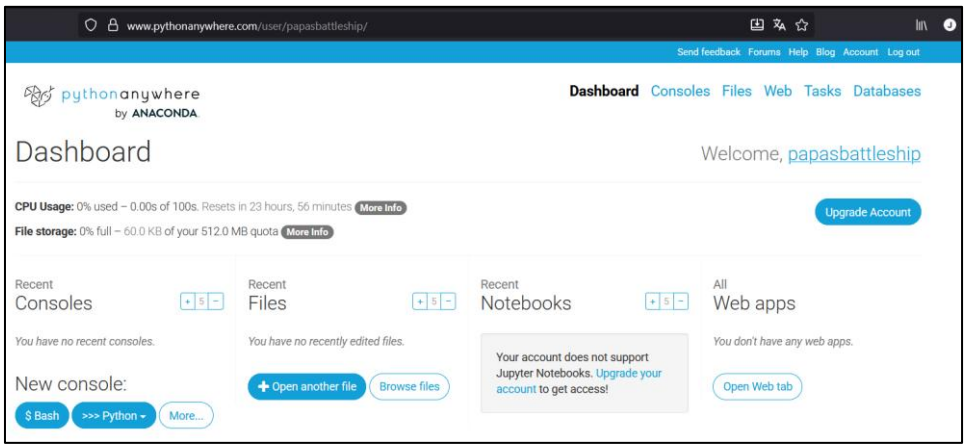




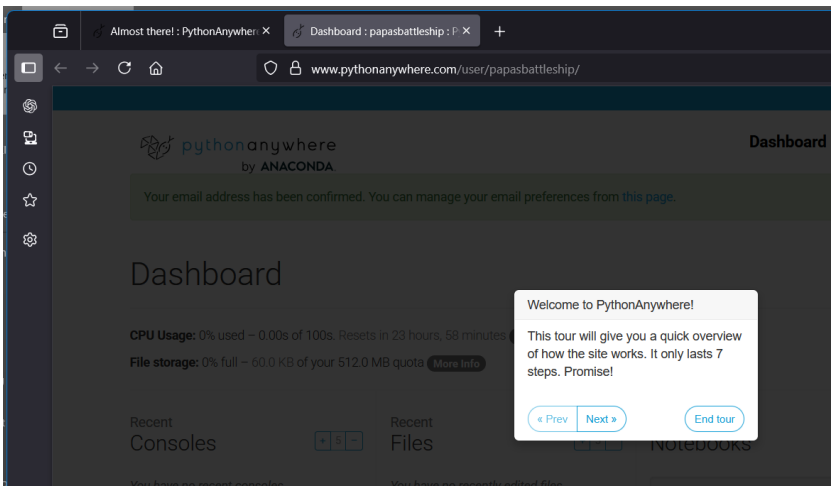
Nach der Anmeldung erhält man eine Mail mit einem Bestätigungslink:



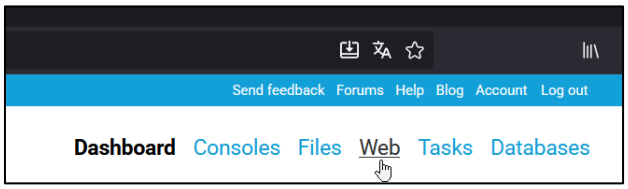
Sobald man das bestätigt hat, landet man im „Dashboard“:



Die angebotene Tour kann man machen, oder auch ablehnen:

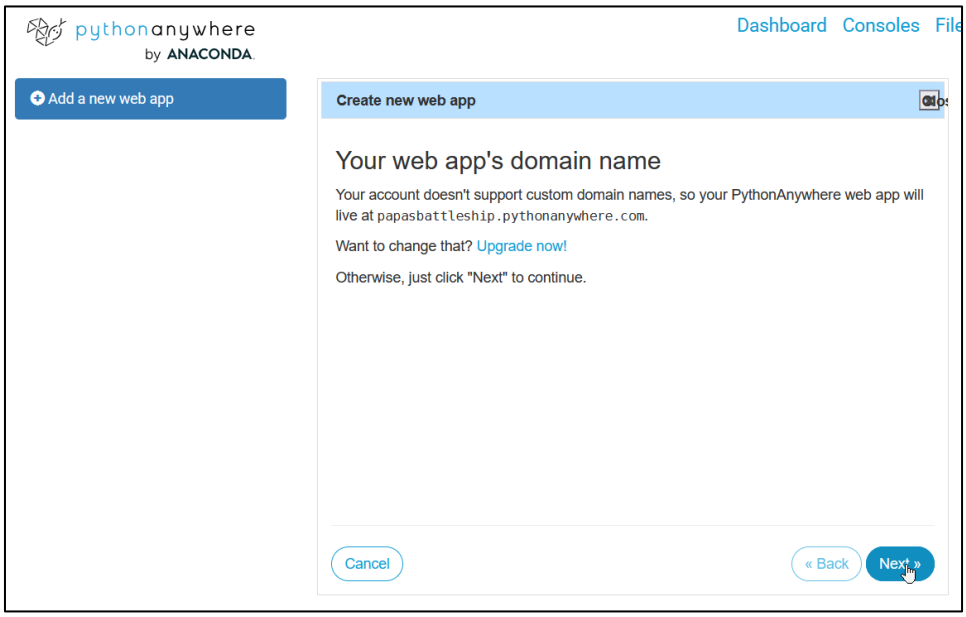


Im Dashboard auf „Web“ oben rechts klicken:

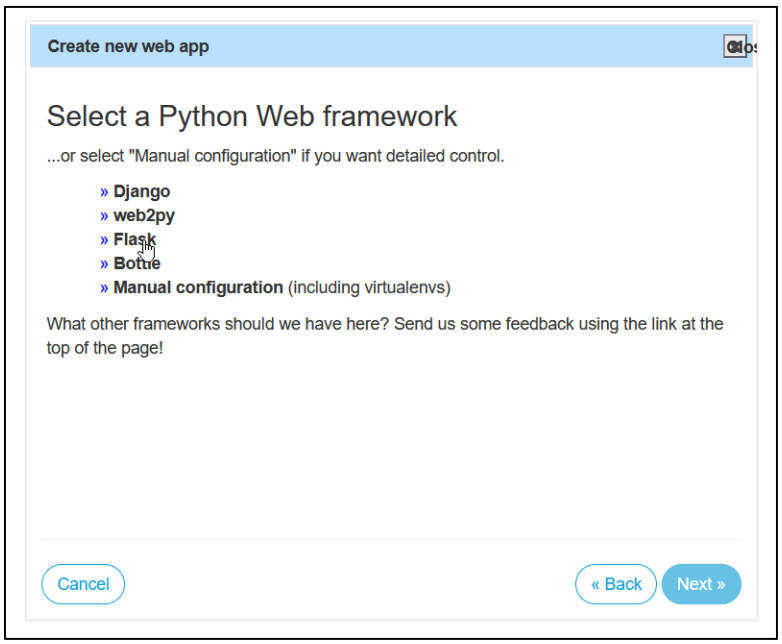


Dann im neuen Fenster „Add a new web app“:

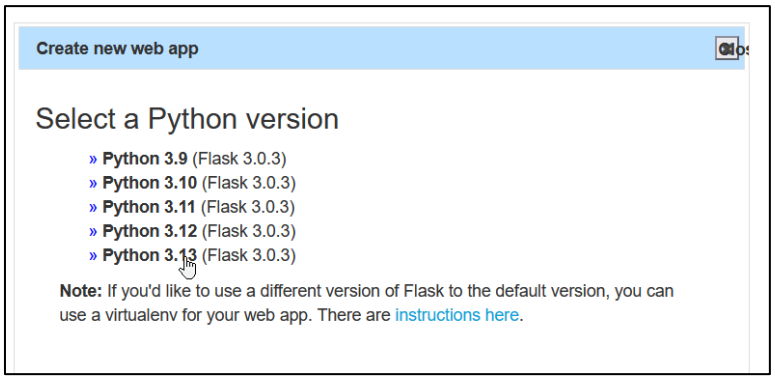
Da wir nicht bezahlt habe, im nächsten Fenster auf „Next“ klicken:



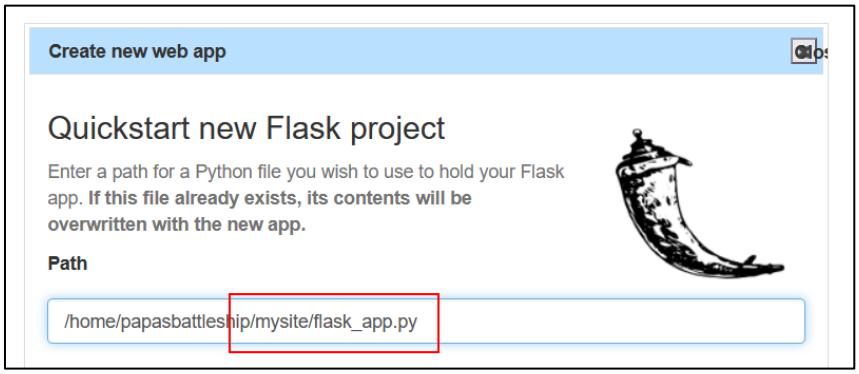
Jetzt müssen wir ein Framework angeben – das ist in unserem Fall „Flask“



Wir werden nach der Python-Version gefragt, da nehmen wir die, die bei uns im Terminal-Fenster erscheint, wenn wir `python --version` angeben:

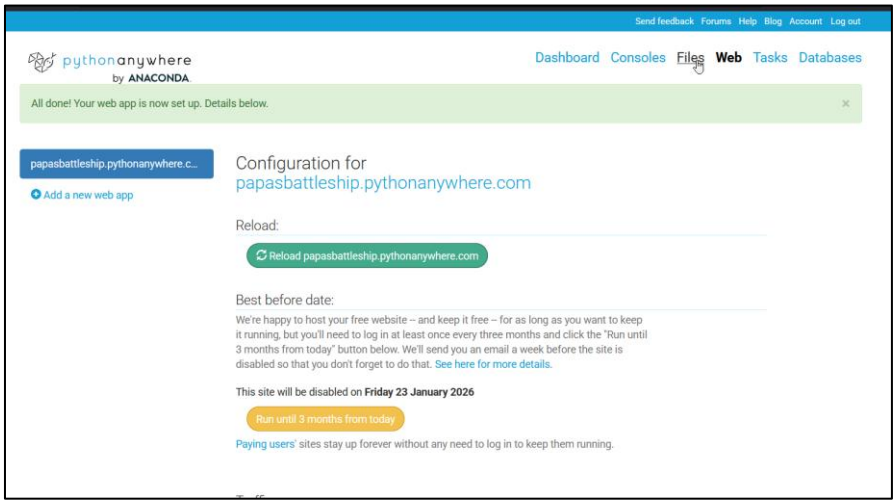


Hier jetzt aufpassen – Standardmäßig wird uns jetzt ein Vorschlag zur Einstiegsseite unterbreitet:

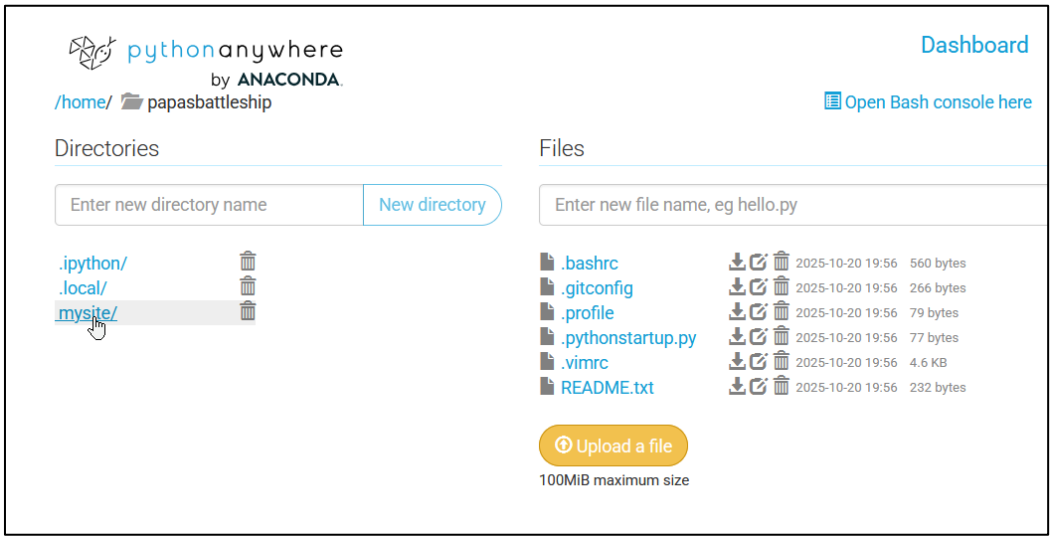


Da unsere App „`app.py`“ heißt, müssen wir das „`flask_`“ löschen.

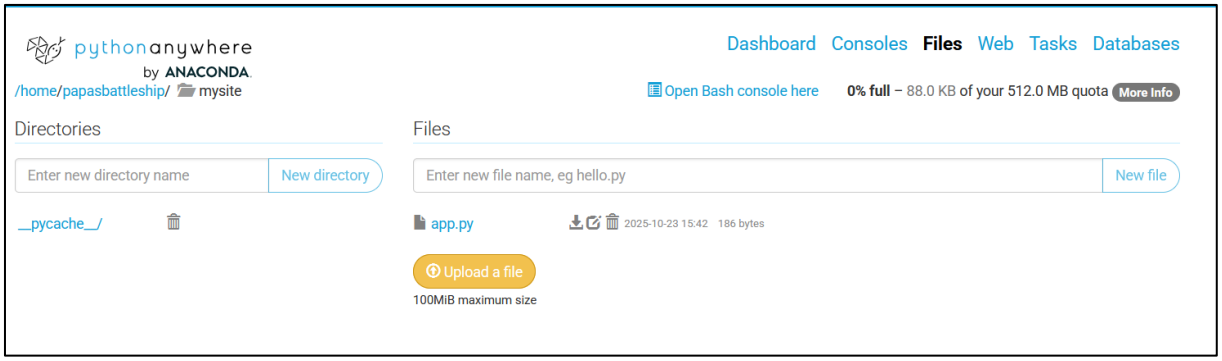
Die Seite wird angelegt, wir haben aber ja noch keinen Inhalt hochgeladen. Damit landen wir auf der Konfiguration. Weiter geht es oben rechts mit „`Files`“:



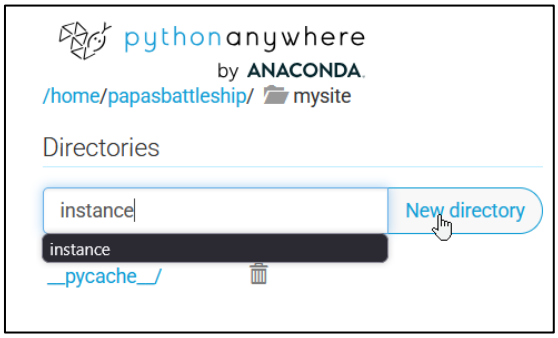
Unsere Dateien müssen alle unterhalb der „`mysite`“-Seite liegen. Um dahin zu gelangen, einfach den Link anklicken:



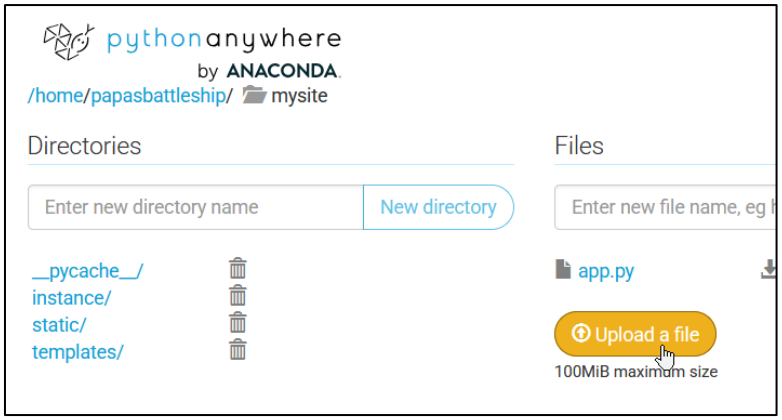
Die erste Datei ist schon angelegt, das ist unsere „app.py“, allerdings noch ohne unseren Inhalt, den müssen wir noch hochladen.



Um einen neuen Ordner anzulegen, einfach den Namen oben links eintragen und „New Directory“ anklicken:

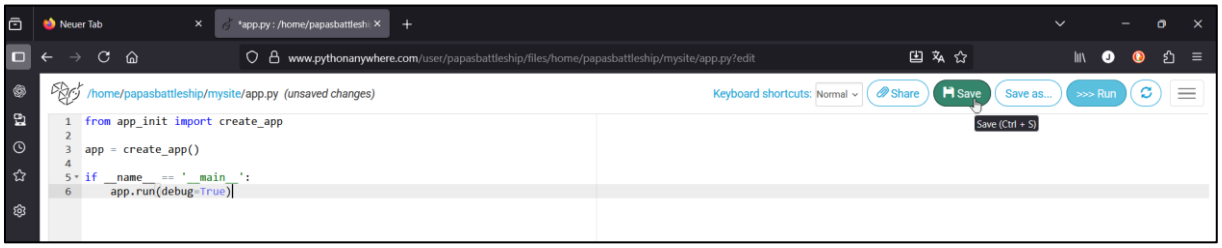


Das machen wir mit unseren 3 Ordnern „instance“, „static“ und „templates“.

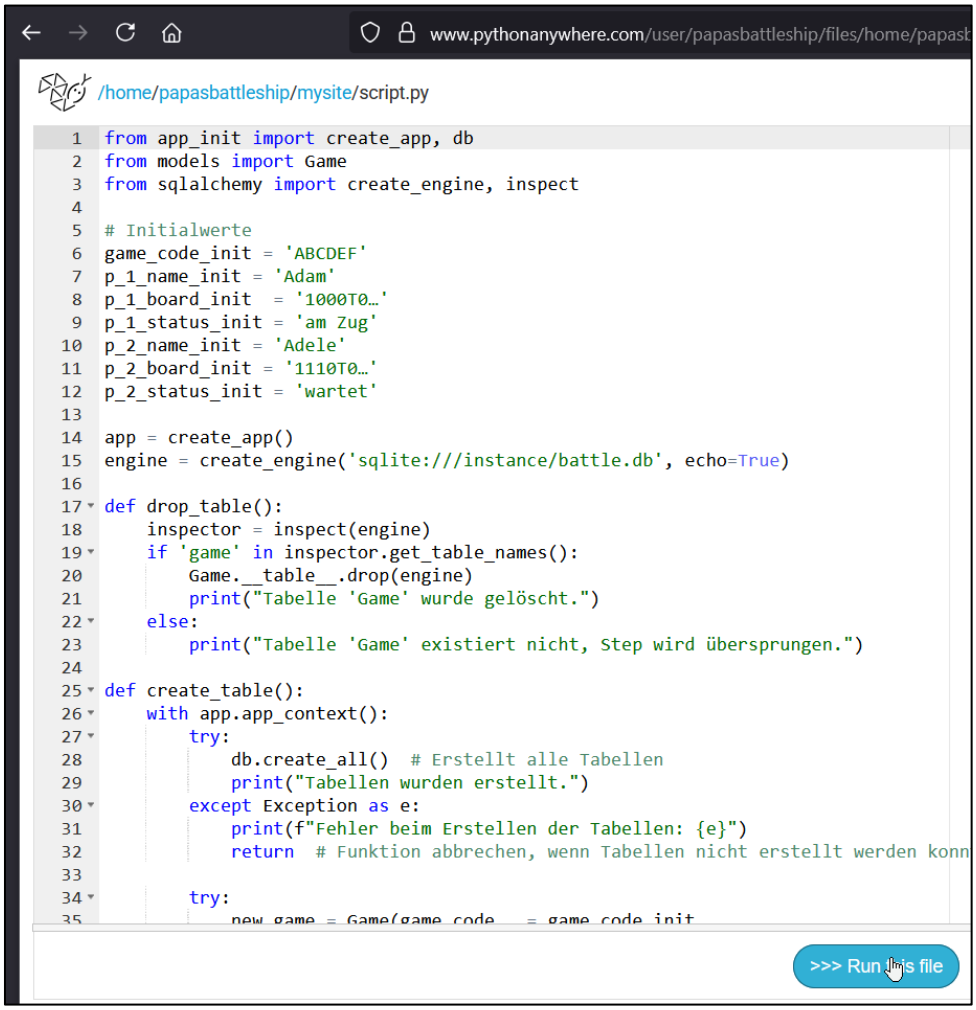


Es gibt wie immer mehrere Möglichkeiten, wie wir die Inhalte auf den Server bekommen. Entweder legen wir sie auf dem Server an und kopieren den Inhalt aus unserem Arbeitsverzeichnis dort hinein, oder wir laden die Files der Reihe nach über den Button „Upload a file“ hoch.

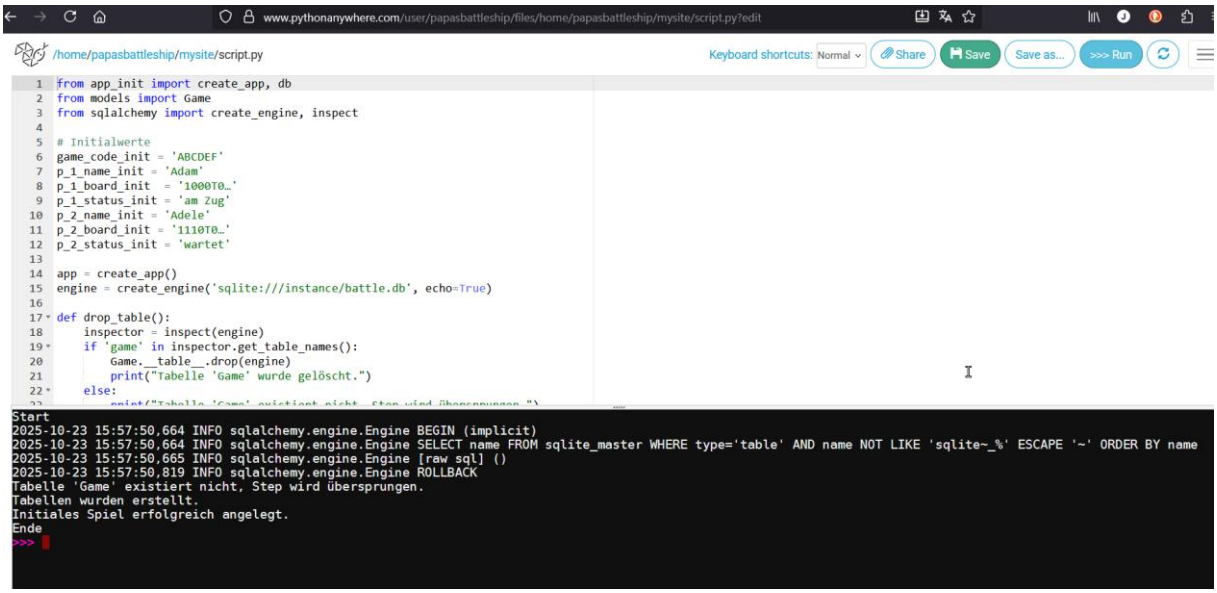
Beim Editieren nicht vergessen den „Save“ Button zu klicken!



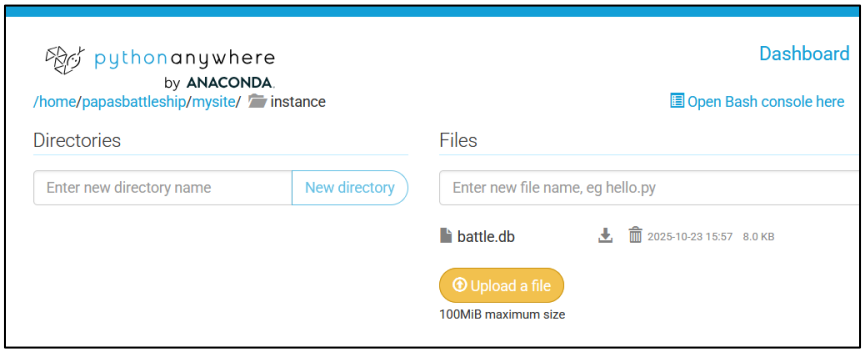
Die Datenbank können wir auch einfach per Upload hochschieben, wir könnten aber auch unser Skript ausprobieren. Dazu unten rechts den Button „Run this file“ klicken:



Dann sollte sich ein Terminal-Fenster öffnen und im besten Fall unsere 3 Hinweistexte kommen:

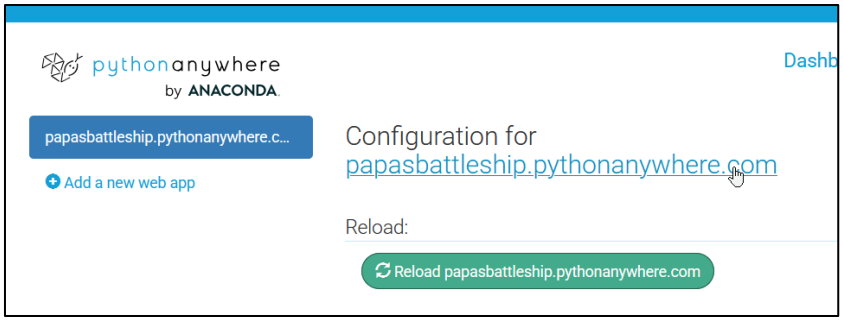


Zur Kontrolle kann man jetzt schauen, ob die `battle.db` angekommen ist, wenn man sie nicht dorthin kopiert hat.

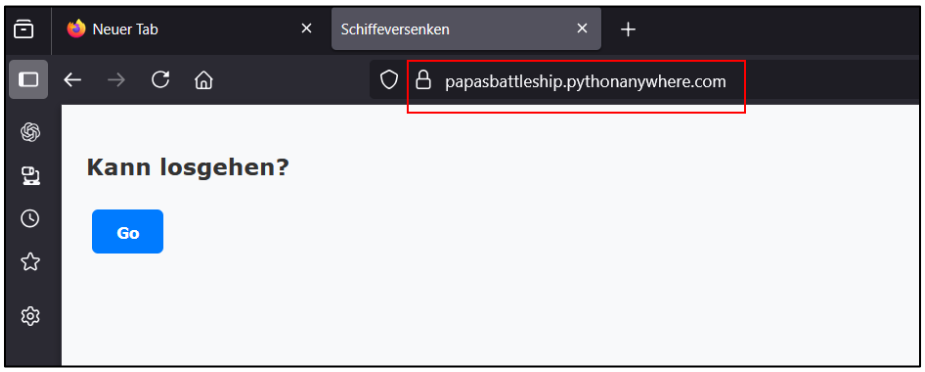


Nachdem wir alle Ordner und Dateien übertragen oder editiert haben, dürfen wir ncht vergessen, die Seite neu zu laden, andernfalls sind unsere Änderungen nur auf dem Server gespeichert, sie werden aber nicht angezeigt.

Dazu in der Config-Seite den Button „Reload papasbattleship.pythonanywhere.com“ klicken:



Nach ein paar Sekunden sollte der Aufruf der Seite unsere index.html anzeigen:

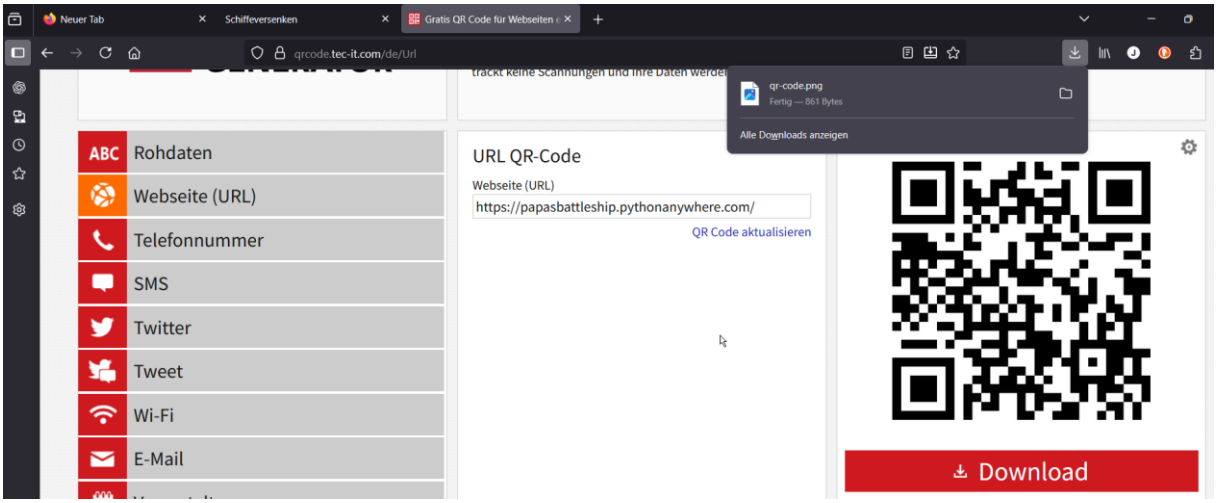


Yes, alles richtig gemacht. Auch hier ruhig einmal ein ganzes Spiel spielen, damit man sieht, dass alles gut geklappt hat.

Damit habt Ihr Eure Version des Spiels in Internet gebracht, BRAVO! Seid stolz auf Euch und lasst alle Eure Freunde daran teilhaben!

Eine letzte Sache noch als kleine Spielerei, wenn Ihr den Link verschickt, könnt Ihr auch vorher online und kostenlos einen QR-Code dafür generieren lassen, das ist dann noch cooler einen QR-Code zu verschicken, als einen Link!

Ich habe das mal für meine Version gemacht, googelt einfach nach „qr generator free“, da werdet Ihr sicher fündig.



5 Abschluss

So, damit bin ich auch schon wieder am Ende des Projekts angelangt. Ich hoffe, es hat Euch ebenso viel Spaß gemacht wie mir. Falls ich irgendwo etwas nicht tief genug oder unverständlich erklärt haben sollte, lasst es mich gerne wissen.

Was gäbe es noch zu tun? Raum für Verbesserungen gibt es immer.

Oben habe ich ja schon angesprochen, dass man eine Statistik ausgeben könnte, wie oft sich welcher Spieler schon angemeldet hat und wie oft der gewonnen oder verloren hat. Ich würde das über eine neue Datenbanktabelle lösen, in der ich den Namen des Spielers, und die beiden Felder `anzahl_siege` und `anzahl_niederlagen` aufnehmen würde. Immer, wenn ein Spiel rum ist, müsste die Datenbanktabelle aktualisiert werden. Das könnte man in der `gameOver`-Route umsetzen.

Was mir aber noch dringender erscheint, im Spiel selbst könnte es passieren, dass ein Spieler nicht mehr verbunden ist. Stand jetzt kann das Spiel kein „Wiederaufsetzten“. Aus meiner Sicht ein unbedingtes Muss, habe ich aber noch nicht entwickelt. Vermutlich würde man das über das Speichern und Abfragen eines Timestamps je Spieler machen. Wenn ein Spieler der den Status „aktiv“ hat, für mehr als z.B. 2 Minuten keine Aktivitäten zeigt, könnte man das Spiel für beide Spieler abbrechen.

Was man auch machen könnte, ist die Schiffe per „Drag&Drop“ auf das Spielfeld zu bringen.

Und wie geht echte Mehrsprachigkeit?

Damit lasse ich Euch mit Eurem neugewonnenen oder aufgefrischten Können wieder alleine.

Ansonsten gilt wie immer, viel Spaß beim Nachbauen und wenn Euch was Tolles einfallen sollte, was man umsetzen könnte, schickt mir gerne eine Mail an papa@papa-programmiert.de, ich würde mich sehr freuen, genau wie auch über eine Rückmeldung zu diesem Dokument.

Viel Spaß weiterhin beim Coden, bleibt neugierig und hartnäckig.

Viele Grüße, Papa