

## WBE-Praktikum 10

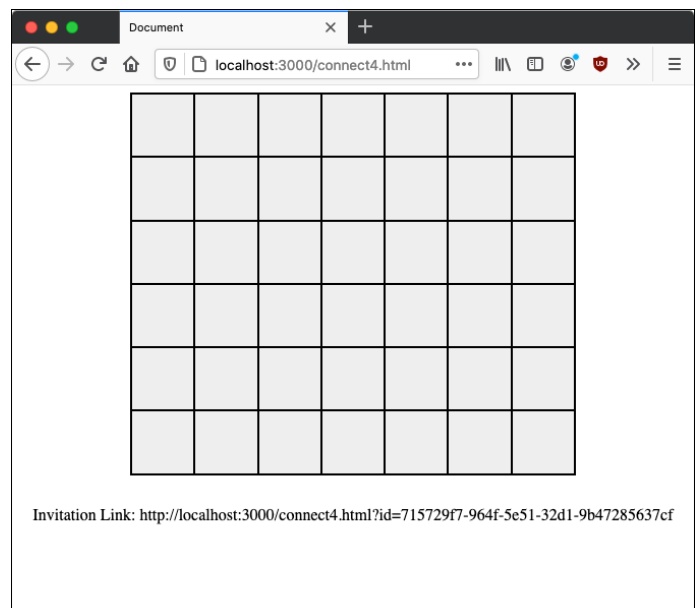
# Spiele im Browser

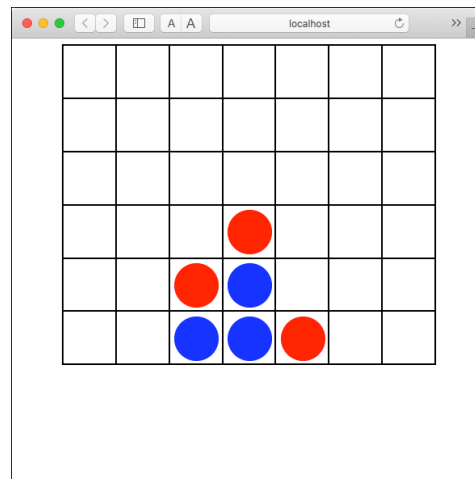
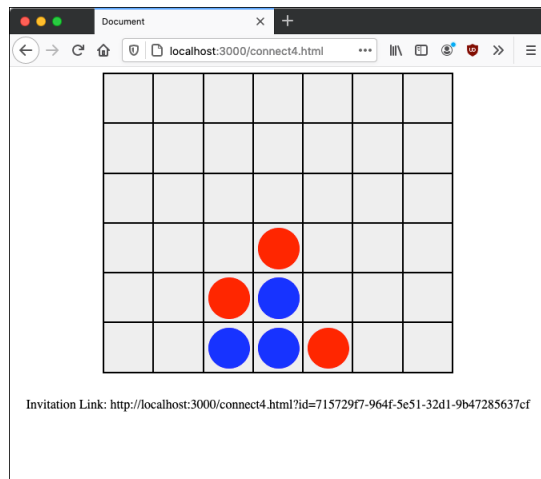
## Aufgabe: «Vier gewinnt» mit Server-Interaktion

Das im letzten Praktikum implementierte Spiel «Vier gewinnt» soll nun so umgebaut werden, dass zwei Spieler in eigenen Browsern gegeneinander spielen können. Die Kommunikation läuft dabei über einen Server. Die Idee ist, dass nach dem Aufruf des Spiels ein Link eingeblendet wird, unter dem ein Mitspieler am Spiel teilnehmen kann (s. Bild).

Damit das nicht zu kompliziert wird, gehen wir folgendermassen vor:

- Als Server verwenden wir unseren REST Service aus Praktikum 6 (Aufgabe 3). Wir speichern einfach den gesamten Spielzustand (Variable *state*) auf dem Server.
- Unser REST-Service speichert beliebige Datenstrukturen unter einer *id*, welche nach einem POST-Request automatisch generiert und zurückgegeben wird. Unter dieser *id* kann dann in der Folge auf die gespeicherten Daten zugegriffen werden.
- Damit wir keine Probleme wegen *Cross Origin Requests* bekommen, laden wir die HTML- und CSS-Dateien vom gleichen Server, der auch die REST-API zur Verfügung stellt. Hinweise zu *Cross Origin* Problemen finden Sie am Ende dieser Praktikumsbeschreibung.
- Wir verwenden keine Echtzeitkommunikation mit dem Server. Stattdessen holen wir in regelmäßigen Abständen (zum Beispiel alle zwei Sekunden) den aktuellen Zustand vom Server. Diese als *Polling* bezeichnete Technik führt natürlich zu kleinen Verzögerungen beim Abgleich des Zustands, ist aber relativ einfach zu implementieren. Ein paar Hinweise zu alternativen Techniken finden Sie am Ende dieser Praktikumsbeschreibung.
- Da im Zustand auch angegeben ist, ob rot oder blau den nächsten Zug hat, kann die Eingabe anhand dieser Information jeweils für einen der Mitspieler gesperrt werden. Mit einem grauen Hintergrund wird angezeigt, dass aktuell keine Eingabe möglich ist (s. Bilder auf der nächsten Seite).





### Aufgaben:

- Zunächst zum Server. Wie schon im früheren Praktikum verwenden wir *Express* für den Server. Um auch statische Dateien wie *connect4.html* und *styles.css* übertragen zu können ist eine zusätzliche Middleware nötig. Ausserdem erlauben wir einen zusätzlichen API-Key *c4game*. Sehen Sie sich *index.html* mit den zugehörigen Änderungen an.
- Installieren Sie die zusätzlich benötigten Module (*npm install*). Wenn Sie den Server nun starten (*node index.js*) sollten Sie das Spiel laden können (*http://localhost:3000/connect4.html*).
- Nun zum Client. Vorläufig ist das Script noch im HTML-Code eingebettet. Die Funktion *initGame* dient zur Initialisierung. Zunächst wird überprüft, ob die URL eine *id* angehängt hat. Ist das der Fall handelt es sich um den eingeladenen Spieler: die eigene Farbe (*session.me*) wird von rot auf blau umgestellt. Falls nicht, muss das Spiel auf dem Server erst noch angelegt werden (*postState*).
- Implementieren Sie die Funktion *postState*: Zunächst wird der Zustand mit POST an den Server geschickt. Dieser antwortet mit einer *id*. Diese wird im Session-Objekt gespeichert. Ausserdem wird sie zur Generierung einer Einladungs-URL verwendet, die auf der Seite unter dem Spielfeld angezeigt wird (*document.body.appendChild*).

Zum Zugriff auf den Server können Sie *fetch* verwenden, welches eine Promise zurückgibt. Beispiel für das Senden von POST-Daten und Auswerten der Antwort, beides JSON:

```
fetch(url, {
  method: 'POST',
  headers: { 'Content-type': 'application/json' },
  body: jsonData
})
.then(response => response.json())
.then(data => { /* use data */ })
```

- Ergänzen Sie die Ereignisbehandlung (*attachEventHandler*) so, dass Mausklicks nur ausgewertet werden, wenn man selber am Zug ist.

- Ergänzen Sie die Ausgabe des Spielfelds (*showBoard*) so, dass die Klasse *inactive* gesetzt oder gelöscht wird, je nachdem ob man selber am Zug ist oder nicht. Hinweis: DOM-Knoten haben ein Attribut *classList* mit Methoden *add* und *remove*.
- Implementieren Sie die Methode *getState*. Sie holt den Zustand vom Server und ersetzt damit den bisherigen Zustand. Anschliessend wird das Spielfeld neu ausgegeben. Diese Methode wird in regelmässigen Intervallen aufgerufen und sorgt dafür, dass beide Spieler den gleichen Spielstand angezeigt bekommen.
- Implementieren Sie die Methode *putState*. Sie sendet den aktuellen Zustand an den Server. Ergänzen Sie die Ereignisbehandlung so, dass *putState* immer nach dem Einfügen eines Spielsteins aufgerufen wird.

Nun sollte alles beieinander sein und das Spiel müsste in zwei Browsern spielbar sein. Probieren Sie es aus. Natürlich könnte das Spiel noch in verschiedener Hinsicht verbessert werden, was aber den Zeitrahmen eines Praktikums sprengen würde. Ein paar Ideen für diejenigen, die irgendwann einmal Zeit und Spass haben, das Projekt weiterzuentwickeln:<sup>1</sup>

- Erkennen, ob ein Spieler gewonnen hat, fehlt immer noch
- Match aus mehreren Spielen und zählen, wer wie oft gewonnen hat
- Spiel aufgeben und neues Spiel starten
- Eingabe von (Spitz-) Namen der Spieler
- Zufällige Auswahl wer den ersten Zug hat
- Liste von Teilnehmern auf dem Server, die nach Spielpartnern suchen
- Animation beim Einfügen von Spielsteinen
- Visuelles Feedback, wenn man in eine Spalte geklickt hat, die bereits voll ist oder wenn man klickt obwohl man nicht am Zug ist

## Ergänzende Hinweise

### Polling

Das regelmässige Abfragen eines Servers, ob neue Informationen vorliegen, wird wie oben beschrieben als **Polling** bezeichnet. Da HTTP-Anfragen vom Client initiiert werden, ist dies zunächst die naheliegende Variante. Nachteile: unnötiger Netzwerkverkehr, unnötige Serveraktivität. Vorteil: einfach zu implementieren. Eine Variante wird als **Long Polling** bezeichnet: vom Client kommt wie gehabt eine HTTP-Anfrage, der Server wartet aber mit der Antwort, bis neue Informationen vorliegen oder ein vordefiniertes Zeitintervall abgelaufen ist. Eine weitere Alternative sind **WebSockets**: hier wird

---

<sup>1</sup> Die Ergebnisse können Sie uns gerne zur Verfügung stellen (auch wenn das Semester dann bereits vorbei ist). Vielleicht ergeben sich daraus Anregungen für kleine Projekte oder weitere Praktikumsaufgaben, bzw. verbesserte Musterlösungen.

eine TCP-Verbindung zwischen Client und Server aufgebaut, über die beide Seiten Informationen senden und empfangen können.

Einige Hinweise zur Vertiefung (nicht Prüfungsstoff):

- Seite zu WebSockets des Mozilla Developer Networks:  
<https://developer.mozilla.org/de/docs/WebSockets>
- Socket.io ist eine Bibliothek zur Echtzeitkommunikation, verwendet WebSockets, Long Polling als Fallback, wenn WebSocket-Verbindung nicht zustande kommt  
<https://socket.io/docs/>
- Faye ist ein „publish-subscribe“ Nachrichtensystem  
<https://faye.jcoglan.com>

### Same Origin Policy

Aus Sicherheitsgründen darf ein Script normalerweise nur mit Ressourcen auf derselben Quelle interagieren, von der es selbst heruntergeladen wurde. Sonst könnte ein Script einer Website, nennen wir sie <http://malicious.attacks/you-have-won.html>, schlimmstenfalls mit anderen Seiten interagieren, in die wir gerade eingeloggt sind (Facebook, Bank). Der Server der anderen Seite kann einen solchen Zugriff jedoch zulassen, indem er mit einem *Access-Control-Allow-Origin*-Header alle oder bestimmte Quellen zulässt. Dies wird als **Cross-Origin Resource Sharing (CORS)** bezeichnet.

- Seite zu CORS des Mozilla Developer Networks:  
<https://developer.mozilla.org/de/docs/Web/HTTP/CORS>