Einführung:

Auf zum Endspurt! Dieses Aufgabenblatt ist für zwei Termine ausgelegt! Nutzen Sie diese Gelegenheit, um alles rund zu bekommen! Am Ende des letzten Termins soll alles laufen und abgenommen werden können.

In diesem Aufgabenblatt beschäftigen Sie sich damit weitere Konzepte in das Verteilte System hinein zu bringen.

Hierbei wird zum einen Heterogenität geschaffen, dass alle Komponenten gleich betrachtet werden und so auch der Client als Service auftritt, um erreichbar zu sein.

Für den Event-Service wird das Subscriber-Model angewendet, so dass der Client nicht ständig selbst darauf achten muss, ob etwas neues passiert ist, sondern notifiziert wird, wenn etwas passiert.

Monopoly ist langweilig, ohne dass man etwas kaufen kann! Somit werden die Broker eingeführt, welche für das Management der Straßen und der Abwicklung von Mieten zuständig sind.

Wie immer sind Banken sehr wichtig. Somit wird dafür gesorgt werden, dass diese „hochverfügbar“ sind. Hierfür ist es notwendig die Services redundant zu halten. Hierfür ist es natürlich notwendig, dass diese Services synchron laufen, dass bei einem Ausfall es zu keinem Datenverlust kommt.
Unterschätzen Sie diese Aufgabe nicht! Es gibt einige Fallstricke und Fälle zu bedenken!

Aufgabe 3.3 A : Die /Brokers

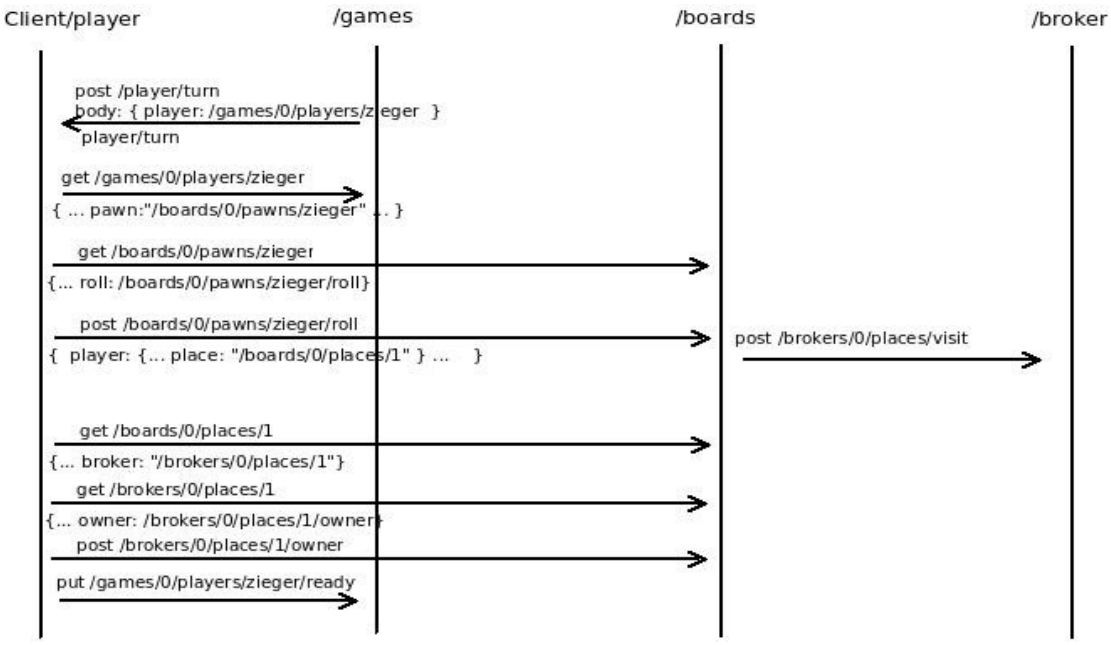
1.) Implementieren Sie den /brokers-Service, so dass

- /boards einen neuen pro Spiel erstellt
  `put /brokers/{gameid}`
- /boards die verfügbaren Grundstrücke registriert mit
  `put /brokers/{gameid}/places/{placeid}`
- /boards bei /brokers Besuche durch Spieler anmeldet
  `post /brokers/{gameid}/places/{placeid}/visit/{playerid}`
- Spieler Grundstücke kaufen können durch
  `post /brokers/{gameid}/places/{placeid}/owner`
- /brokers bei fälliger Miete oder Erwerb es der Bank meldet

2.) **Verteilte Transaktion:** Achten Sie darauf, dass der „Erwerb von Grundstücken“ als eine Transaktion gesehen wird. Insbesondere muss darauf geachtet werden, dass „Besitzer eintragen“ und „Geldtransfer“ immer entweder beides geschieht oder nichts. Hierzu können Sie das Transaktionsinterface der Bank benutzen, müssen allerdings bedenken, dass es sich nun um eine verteilte Transaktion handelt

Aufgabe 3.4 A: Eine Runde für alle!

Das System ist nun so weit fortgeschritten, dass ein fast vollständiger Rundenablauf möglich sein sollte. Überprüfen Sie und erweitern Sie gegebenfalls den Client und die Services so, dass ein Ablauf wie folgendes Beispiel möglich ist. Achten Sie darauf, dass möglichst HATEOAS benutzt wird und nur mit Links zu anderen Resourcen gearbeitet wird (defakto keine String-Konkatenation auf der Client-Seite). Denken Sie an den Location-Header!



Aufgabe 3.1 B: Client-as-a-Service

Implementieren Sie den Spieler/Client als /client-Service. Dies ermöglicht eine homogene bidirektionale Kommunikation zwischen Spieler/Client und Services. Der /client-Service fällt etwas aus dem Schema der anderen Services, so dass dieser nicht zentral sein muss. Entweder gibt es einen Service, bei dem sich der Client mit konstanter Verbindung anmeldet, oder er baut einen eigenen Service auf. Die URL des Services wird mit dem Spieler registriert. Anforderung ist schlicht, dass der Client direkt Nachrichten zugestellt bekommen kann, ohne Polling betreiben zu müssen bei verschiedenen Diensten.

Implementieren Sie den Service so, dass

- der Spieler benachrichtigt wird, wenn er am Zug ist per
  `post /client/turn`
- dem Spieler Ereignisse zugestellt werden können mit
  `post /client/event`

Aufgabe 3.2 B: /events subscription

Das Subscriber-Modell ist eine weit verbreitete Methode, um Ereignisse in einem Kreis von Interessenten zu verteilen. Hierbei kommt der /client-Service zum Einsatz, so dass Interessenten sich mit der URL zum /client-Service einschreiben können und direkt benachrichtigt werden, wenn neue Events vorliegen.

1.) Erweitern Sie deshalb den /events-Service, so dass

- Interessenten sich einschreiben können mittels
  `post /events/subscriptions`
- alle Interessenten benachrichtigt werden über neue Events per
  `post {uri of the subscription}`

2.) Machen Sie sich bei der Implementierung Gedanken welche Kommunikations-Muster Sie an welcher Stelle verwenden, um Deadlocks oder zu lange Requests zu vermeiden!

Erstellen Sie ein kurzes Beispiel Diagramm über den Ablauf ab Erzeugung eines Events, bis zur Reaktion des Client. Sie dürfen die genutzten Services frei wählen, allerdings muss eine Reaktion des Client erfolgen (wenn auch nur theoretisch)!

Aufgabe 3.5 A&B: Redundante /banks

Dies ist eine Team-Aufgabe! Denkt gemeinsam, entwickelt gemeinsam! Es ist komplexer als man auf den ersten blick glaubt.

Die Banken sind ein essenzieller Bestandteil und von besonderer Wichtigkeit. Deshalb soll hier eine **Ausfalltolleranz** geschaffen werden, welche über **Replikation** erreicht wird. Hierbei wird der Service repliziert und über einen Proxy angesprochen.

Wichtig hierbei ist die **Synchonisierung** und **Konsistenz** der Daten. Achten Sie darauf, dass wenn einem Kunden die Überweisung zugesichert wurde, dass alle Instanzen von /banks die Änderung auch übernommen haben. Überlegen Sie ob oder wie Lastverteilung hierbei erreicht wird.

Es muss hierfür keine gemein gültige API benutzt werden – die Synchronisierung erfolgt nur unter homogenen System.

Nutzen Sie als Proxy **nginx**. Dies ist ein leichtgewichtiger Reverse-Proxy, welcher einfach zu Konfigurieren ist und ein einfaches Docker-Image anbietet. Im Gegensatz zur Alternaive HAProxy, überstützt nginx Namensauflösung (beim Start), so dass Sie nicht die IP der Container vorher wissen müssen, sondern mit den Namen Arbeiten können (<user>_<directory>).

Es ist Ihnen frei überlassen, welche Algorithmen bzw. Techniken Sie verwenden, um die Ausfalltolleranz inklusive Synchronisation zu erreichen.

Tipp: Wenn Sie eine zusätzliche zentrale einzelne Komponente einführen, sinkt die Verfügbarkeit effektiv!!

Wir nehmen an, dass der **nginx** als Proxy NIE ausfällt. Es ist also kein Problem, dass dieser nicht redundant gehalten wird. In echten Umgebungen arbeitet man mit zusätzlichen IP einträgen im DNS oder verschiebbaren virtuellen IPs.