Vorbereitung

Die Sprache: Java! Oder etwas anderes…

Eine Stärke von REST ist es ja gerade, unabhängig von einer bestimmten Programmiersprachen zu sein. Es ist richtig, dass es für Java (auch durch uns) mehr Support gibt, aber Ihre anderen (Lieblings-) Sprachen werden auch akzeptiert – solange es funktioniert. Es liegt also allein an Ihnen, uns davon zu überzeugen!

Außerdem müssen Sie dann einen lauffähigen Docker-Container vorlegen (siehe Allgemeine Hinweise), welcher Ihre Anwendung ausführt.

Für Java können Sie z.B. folgendes nutzen:

- **Maven2** Projekt aufsetzen
  Maven ist ein sogenannter „build and dependency manager“, der automatisch die angegebenen Dependencies aus den Repositories zieht.
  https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

- **Spark Rest Framework** besorgen und Dependency eintragen
  Spark ist ein simples Rest-Framework auf Basis von Jetty und mit Maven einfach zu integrieren. Vorteil für Sie: Sehr flache Lernkurve!
  http://sparkjava.com

- **Gson** besorgen und Dependency eintragen
  Gson ist eine Library zum Handhaben von Json

Alle angegebenen Libraries und Frameworks sind nur Empfehlungen und damit Vorschläge. Wie schon bei der Sprache an sich gilt: Überzeugen Sie uns, dass alles klappt!

Aber etwas müssen Sie sich unbedingt besorgen! … eine REST-Test-Erweiterung für Ihren Browser, z.B. für Firefox **REST Easy**

Viel Erfolg!

Allgemeine Hinweise

Docker
Die von Ihnen entwickelten Services werden in Docker-Containern laufen. Wer in Java entwickelt und ein Jar (+ Dependencies) produziert, muss sich mit den Details nicht beschäftigen. Es wird ein Upload angeboten, der das Jar in einen Docker-Container lädt und den Service startet. Der Prozess ist ausführlich beschrieben in der Anleitung zur Abgabe.

Die Container sind von außen über einen HTTP-Proxy mittels der Container-IP und dem Container-Port erreichbar (Schema für die URL: vsdocker/cnt/<ip>/<port>).

Untereinander innerhalb der Docker-Netzwerks können die Container frei kommunizieren, d.h. alle Ports sind direkt erreichbar! Das bedeutet aber nicht, dass die Container bzw. Ports auch aus dem Internet direkt erreicht werden können. Deswegen gibt es mit OPENVPN einen geschützten Zugang, über den Sie bzw. Ihre Anwendungen einen direkten Zugriff in das Container-Netz erlangen, andere Internet-Benutzer aber eben nicht!

Wer sich etwas spezielles wünscht, wie z.B. eine andere Programmiersprache oder bestimmte Pakete als Voraussetzung, muss dafür einen eigenen Docker-Container bauen. Weitere Hinweise für die Erstellung, sowie Verwaltung von Containern finden Sie in dem Beiheft „VSAI-sose2016-Docker“, mit ausführlicher Ausführung wie mit Containern in VS umzugehen ist.

Gruppen
Die Aufgaben sind in 4er Gruppen zu bearbeiten, zusammengesetzt aus jeweils zwei 2er Teams.

Aufgaben gekennzeichnet mit A und B werden nur von jeweils einem der beiden Teams gefordert. Aufgaben ohne A oder B sind von allen Teams zu bearbeiten.

Es ist erforderlich, dass sich die beiden Teams einer Gruppe stark untereinander austauschen, da nur gemeinsam ein Gesamtsystem entstehen kann.

API-Spezifikation
Die Spezifikationen für die Services finden Sie in der OwnCloud. Dieses Aufgabenblatt führt noch die meisten Spezifikationen auf. Ab dem nächsten Aufgabenblatt wird erwartet, dass die Spezifikation aus der OwnCloud benutzt wird.

Da immer weiter an dieser Spezifikation gearbeitet wird, sind Vorschläge gerne gesehen. Sollten diese so wichtig sein, dass noch innerhalb dieses Semesterdurchlaufes etwas geändert werden muss, wird die Spezifikation angepasst und eine Mail über den Verteiler geschickt.

Aufgabe 1.2: Project RESTopoly

Das große Ziel ist es, Monopoly als verteilte Anwendung zu implementieren. Endgültig wird es eine Anwendung aus REST-Services werden. Sie werden dabei versuchen, so viele Spielelemente wie möglich umzusetzen, nicht alles ist verpflichtend, aber mit den anderen Elementen macht es mehr Spaß.

Bei RESTopoly werden die Entitäten als eigenständige Services mit REST-Interface implementiert.

Aufgabe 1.2.1 : Frag den Service!

In dem Docker-Netzwerk läuft bereits ein ganz besonderer Service: Ein Verzeichnis-Dienst aka Yellow Pages. Dieser wird benutzt, um weitere Services zu finden. Damit Ihre Services gefunden werden können, ist es notwendig, dass Sie diese sich selbst in das Verzeichnis eintragen lassen, wenn sie gestartet werden. Der YellowPages ist zu finden unter yellow:4567 – solange Sie im VPN des Docker-Netzwerkes sind!

Alle Ihre geschriebenen Services müssen sich bei dem Yellow Pages anmelden. Angemeldete Services **MÜSSEN** unter der angegebenen URI per GET erreichbar sein und einen Status 200 zurück geben, oder alternativ unter http://host:port/api die API-Definition zurück geben mit Status 200. Dies wird benutzt zur Status-Kontrolle.

1.) Nutzen Sie das OpenVPN, um in das Docker-Netwerk zu gelangen

2.) Nutzen Sie die Yellow-API, um Services zu lesen und zunächst einmal einen Fake-Service einzutragen

Aufgabe 1.2.2 A: Die Würfel – REST /dice

1.) Implementieren Sie einen Würfel als REST-Service. Dieser Service muss das folgende Interface erfüllen (in RAML)

```
/dice:
  get:
    description: Gives you a single dice roll
    responses:
      200:
        body:
          application/json:
            schema: |
                      { "type": "object",
                          "properties": {
                            "number": {"type": "int"},
                            "required": true
```

```
                        },
                      }
          example: '{ "number": 4 }'
```

2.) Erweitern Sie den Dice-Service, so dass diesem mitgeteilt werden kann, welcher Spieler würfelt und für welches Spiel. Realisieren Sie dies über zwei queryParameter names „player“ und „game“. Beide Parameter sollen eine URIs akzeptieren als Werte.

Aufgabe 1.2.3 A: Benutzer registrieren - /users

Implementieren Sie einen Service, bei dem sich Benutzer registrieren können. Für den Service wird folgendes Interface benutzt:

```
/users:
  description: "The users service registers users of the system"
  get:
    description: "Returns list of URIs of player resources"
    responses:
      200:
        body:
          application/json:
            example: |
              { "users": ["/users/mario"] }
  post:
    description: "Registers a new player with the system"
    body:
      application/json:
        example: |
          {"name":"Mario",
            "uri":"http://somehost:4567/client/mario"}
  /{userid}:
      get:
        description: Returns the state of the player resource
        responses:
          200:
            body:
              application/json:
                example: |
                  {"id":"/user/mario","name":"Mario",
                   "uri":"http://localhost:4567/client/mario"}
      put:
        description: Registers or changes the user/player
        queryParameters:
          name:
            displayName: Playername
            type: string
```

```
          description: Name of the player
          example: Mario
        uri:
          displayName: Clienturi
          type: string
          description: uri of the client service
          example: http://localhost:4567/client/mario
    delete:
      description: "Unregisters the user"
```

Aufgabe 1.2.4 B: API Spezifikation

RAML ist eine weit verbreitete Spezifikationssprache für REST-Interfaces. Bevor etwas implementiert werden kann, braucht es eine hinreichende Spezifikation, gerade wenn verteilte Teams verteilte Anwendungen entwickeln!

Erstellen Sie eine sinnvolle API-Spezifikation in RAML (http://raml.org/) für einen frei wählbaren Service aus der Liste im Sinne von RESTopoly:

/games	– Meta-Ebene des Spieles, verwaltet Spieler und Runden
/boards	– Spielbrett, verwaltet Platzierungen
/banks	– Bank, die alle Geldangelegenheiten handhabt
/brokers	– Verwaltung von Grundstücken und Mieten
/player	– Der Spieler selbst
/decks	– Die Kartenstapel für Ereignis- und Gesellschaftskarten
/jail	– das Gefängnis

Dabei sollen mindestens zwei Methoden verwendet werden, mindestens eine Antwort und eine Parameterübergabe spezifiziert werden.

Aufgabe 1.2.5 B: /events manager

Damit alle Spieler auf dem Laufenden sind, gibt es einen /events-Service, bei dem alle Events abgelegt werden, die innerhalb des Spieles geschehen. Andere Spieler/Clients können diese Events benutzen, um Informationen anzuzeigen, oder auf Geschehnisse zu reagieren.

1.) Implementieren Sie den /events-Service mit folgender Spezifikation:

```
schemas:
  - event:  |
      {
        "type": "object",
        "$schema": "http://json-schema.org/draft-03/schema",
```

```
        "id": "event",
        "properties": {
          "id": { "type":"string", "required":true,
                  "description":"the url to the event on
                   the event server"},
          "game": {"type":"string", "required":true,
                   "description":"the uri of the game this
                                  event belongs to"},
          "type": { "type": "string", "required": true ,
                    "description":"internal type of the event
          (e.g bank transfer, rent, got to jail, estate transfer)" },
          "name": { "type": "string", "required": true,
                "description":"human readable name for this event"  },
          "reason": { "type": "string", "required": true,
               "description":"a description why this event occured"  },
          "resource": {"type": "string",
               "description": "the uri of the resource related to this
event where more information may be found (e.g. an uri to a transfer or
similar)" },
          "player": { "type": "string",
                "description": "The uri of the player triggering it" },
          "time": { "type": "string",
                "description":"A timestamp when this event
                               was given to the events service"}
        }
      }
/events:
    type: { collection: { schema: "event", "getexample":"",
"postexample":"" } }
    post:
      description: adds a new event to the collection of events
    get:
      description: |
             returns all matching events according
             to query parameters (regex)
      queryParameters:
        game: {"description":"a regex matching for game uri"}
        type: {"description":"a regex matching for event type"}
        name: {"description":"a regex matching for the readable name"}
        reason: {"description":"a regex matching for the reason"}
        resource: {"description":"a regex matching for the uri
                                  referenced by this event"}
        player: {"description":"a regex matching for the player
                                 uri issuing the event"}
    delete:
      description: removes all matching events
```

```
    queryParameters: (same as get)
  /{eventid}:
    get:
      description: gets the event details
```

Hinweis: Derzeit können noch nicht alle Felder sinnvoll ausgefüllt werden…

2.) Passen Sie den /dice-Service so an, dass dieser beim Würfeln einen Event erstellt.