Hochschule RheinMain
Fachbereich DCSM - Informatik
Prof. Dr. Georg Hinkel

Verteilte Systeme

SS 2024
LV 4132

Übungsblatt 7
Praktische Übungen
Abgabe: 16.06.2024, 23:59 Uhr MESZ

Aufgabe 7.1 (Projekt „Hamsterasyl mit RESTFul-Webservice“, 15 Punkte):

Nachdem das westhessische Hamsterverwahrungsunternehmen die RPC-Implementierung getestet und produktiv eingesetzt hat, um sowohl die alten Hasen im Unternehmen, die den Consolen-Client bevorzugten, wie auch die jungen Grünschäbel zufrieden zustellen, die irgendwas mit Java wollten, wurde eine neue Geschäftsleitung berufen. Diese tauschte als erstes die Leitung der IT-Abteilung aus, da dieses offensichtlich zu viel Geld für irgendwelchen Firlefanz ausgibt. Schließlich weiß jeder, der regelmäßig Zeit auf Golfplätzen und First-Class Flughafenlounges verbringt, dass alles REST sein muss!

Die Mitarbeiter der IT-Abteilung haben jetzt die Wahl entweder über Weihnachten eine REST-Lösung zu entwickeln und alle Mitarbeiter umzuschulen, oder sich einen neuen Arbeitgeber suchen zu dürfen (in den Besprechungen fällt in letzter Zeit sehr häufig das Schlagwort „Outsourcing“).

RESTFul-Hamster-Service Zusammenfassung

Aufgabe:

- Implementieren Sie einen RESTFul-Webservice. Die benötigten Bibliotheken sind bereits im Projekt integriert. Die vorgesehene REST-Schnittstelle finden Sie weiter unten auf dem Übungsblatt. Diese Schnittstelle implementiert kein HATEOAS.
- Implementieren Sie außerdem einen REST-Client, der auf Ihren Server arbeitet. Sie finden in der Vorlage bereits wieder eine Vorlage für den Client, der bereits die Kommandozeilenparameter parst und entsprechende Methoden aufruft.

Schnittstelle

Implementieren Sie die folgenden Endpunkte:

- `GET http://localhost:<port>/hamster`
  Liefert die derzeit verwalteten Hamster

Methode `GET`

Query-Parameter Mit dem Parameter *name* kann das Ergebnis auf Hamster eines bestimmten Namens eingegrenzt werden

Body wird ignoriert

Response JSON

```
[{
    "name": <string>,
    "owner": <string>,
    "treats": <integer>,
    "price": <integer>
}]
```

- `GET http://localhost:<port>/hamster/<owner>`
  Liefert die derzeit verwalteten Hamster des angegebenen Besitzers

Methode `GET`

Query-Parameter Mit dem Parameter *name* kann das Ergebnis auf Hamster eines bestimmten Namens eingegrenzt werden

Body wird ignoriert

Response JSON

```
[{
    "name": <string>,
    "owner": <string>,
    "treats": <integer>,
    "price": <integer>
}]
```

- `POST http://localhost:<port>/hamster`
  Fügt einen neuen Hamster hinzu

Methode `POST`

Body JSON

```
{
    "name": <string>,
    "owner": <string>,
    "treats": <integer>
}
```

Response JSON

```
{
    "state": "http://localhost:<port>/hamster/<owner>/<name>"
}
```

- `GET http://localhost:<port>/hamster/<owner>/<name>`
  Zeigt Detailinformation zu dem angegebenem Hamster

Methode `GET`

Body wird ignoriert

Response JSON

```
{
    "name": <string>,
    "owner": <string>,
    "treats": <integer>,
    "turns": <integer>,
    "cost": <integer>
}
```

- `POST http://localhost:<port>/hamster/<owner>/<name>`
  Füttert dem angegebenem Hamster Leckerli

  Methode `POST`

  Pfad `http://localhost:<port>/hamster/<owner>/<name>`

  Body JSON

  ```
  {
      "treats": <integer>
  }
  ```

  Response JSON

  ```
  {
      "treats": <integer>
  }
  ```

- `DELETE http://localhost:<port>/hamster/<owner>`
  Holt alle Hamster des angegebenen Besitzers ab, entfernt alle Datensätze und gibt die angesammelten Kosten zurück

  Methode `DELETE`

  Body wird ignoriert

  Response JSON

  ```
  {
      "price": <integer>
  }
  ```

- *Fehlerfälle:*
  Verwenden Sie die Standard-Statuscodes von HTTP um Fehler anzuzeigen. In diesen Fällen können Sie als Response einfach nur den Text der Fehlermeldung zurückliefern.

Test

Zum Testen steht Ihnen wieder im Verzeichnis `HamsterRPC_Client` eine Java-Testsuite zur Verfügung.

Beispielausgaben für den Client:

- hamster list

```
Owner   Name    Price   treats left
schmidt baggins 17€      17
mueller mueller 17€      22
meier   meier   17€      0
```

- hamster list --owner mueller

```
Owner   Name    Price   treats left
mueller mueller 17€      22
```

- hamster add schmidt baggins 17

```
(HamsterID)
```

- hamster add schmidt baggins

```
(HamsterID)
```

- hamster feed schmidt baggins 3

```
14
```

- hamster state mueller mueller

```
mueller's hamster mueller has done 42 hamster wheel revolutions,
and has 22 treats left in store. Current price is 18€
```

- hamster bill meier

```
17
```

Hinweise C

Für dieses Übungsblatt steht kein Template für die Programmiersprache C zur Verfügung. Sie müssen dieses Übungsblatt daher in Java oder C# bestreiten.

Hinweise Java

Das Template für dieses Übungsblatt verwendet Spring Boot 3, sowohl für den Client als auch für den Server. Da das Testframework die Ausgabe des Clients ausliest, ist allerdings der übliche Logging-Mechanismus von Spring für den Client deaktiviert. Leider ist die geläufige OpenAPI-Implementierung für Spring (namens Spring Fox) derzeit inkompatibel mit Spring Boot 3 und den neueren Versionen von Spring Boot 2. Eine SwaggerUI-Schnittstelle steht Ihnen daher leider nicht zur Verfügung.

Hinweise C#

Das Template für dieses Übungsblatt verwendet ASP.NET Core. Dies erfordert, dass Sie zur Laufzeit nicht nur das .NET SDK sondern auch die Runtime für ASP.NET Core installiert haben müssen (diese ist im blanken SDK nicht enthalten). Für ASP.NET Core wird üblicherweise die OpenAPI-Implementierung Swashbuckle verwendet, die in der Vorlage schon eingebunden ist. Sie können daher unter der Adresse *http://localhost/swagger* die generierte SwaggerUI für die von Ihnen entworfene Schnittstelle ansehen und Ihren Server damit testen. Weiterhin erlaubt es Ihnen die JSON-Beschreibung des Webservers, einen typisierten Client zu generieren. Für die Generierung können Sie beispielsweise NSwagStudio verwenden (`https://github.com/RicoSuter/NSwag/wiki/NSwagStudio`).

Aufgabe 7.2 (Resilienz, 5 Punkte):

Beobachtungen haben gezeigt, dass Hamster von Natur aus gefräßige Wesen sind und auf Leckerli im Normalfall immer ansprechen. Eine wichtige Ausnahme ist allerdings, wenn es den Hamstern nicht gut geht. Da dem westhessischen Hamsterverwahrungsunternehmen schon aus finanziellen Gesichtspunkten am Wohlergehen der behüteten Hamster gelegen ist (die Zahlungsmoral der Kunden lässt zu wünschen übrig, wenn sie feststellen, dass die Hamster während der Verwahrung erkrankt sind), wurde die Hamsterlib um eine Funktionalität erweitert, die bei der Gabe von Leckerli prüft, ob die Hamster auch auf das Leckerli reagieren. Es ist daher wichtig, eventuelle Krankheiten der Tiere frühzeitig und automatisiert zu erkennen, um rechtzeitig einen Veterinär zu rufen, der sich um das Tier kümmern kann (die Behandlung des Tieres wird dem Kunden dann separat in Rechnung gestellt).

Das Problem ist nun, dass Hamster ja Lebewesen sind und sich manchmal unvorhersehbar verhalten, beispielsweise Leckerli verweigern auch wenn sie völlig gesund sind. Man hat außerdem festgestellt, dass Kunden sich weigern, in einem solchen Fall die Rechnung für einen Tiermediziner zu bezahlen, wenn dieser nur die einwandfreie Gesundheit der Hamster feststellt (die Kunden meinten in diesen Fällen, sie wären davon ausgegangen). Außerdem wollen die Kunden nicht jedes mal wenn der Hamster sein Leckerli nicht essen will gleich eine Fehlermeldung sehen, stattdessen wäre es wünschenswert, wenn der Server selbstständig versuchen würde, einen fehlgeschlagenen Fütterungsversuch zu wiederholen.

Die Tiermediziner waren von den Einsätzen bei denen sie nur die Gesundheit der Hamster festgestellt haben auch wenig angetan (meinten, sie hätten auch noch andere Tiere, um die sie sich kümmern müssen) und haben angemerkt, erst wenn ein Hamster innerhalb von eines ganzen Tages mindestens zehn Leckerli verweigert, könne man davon ausgehen, dass er ernsthaft krank sei.

Simulation der Hamster

Es wurde ausdrücklich verboten, für Test- und Entwicklungszwecke gezielt Hamster zu vergiften, um die Erkennung von Krankheiten zu testen. Stattdessen wurde die Hamsterlib bereits so angepasst, dass die Prüfung, ob die Leckerli angenommen worden sind hinter einer Schnittstelle gekapselt, für die die Hamsterlib eine Implementierung anbietet, die Krankheiten simuliert[1].

Transparente Wiederholung

Ihre erste Aufgabe ist es nun, Ihren Server aus Aufgabe 7.1 dahingehend zu erweitern, dass der Server fehlgeschlagene Versuche einen Hamster zu füttern wiederholt. Erst wenn ein Hamster drei mal ein Leckerli abweist, sollten Sie die Fütterungsversuche aufgeben und dem Client einen Fehler schicken. Zwischen den Fütterungsversuchen sollten Sie allerdings eine gewisse Zeit warten. Da sich die Tiermediziner was die Länge dieser Zeitspanne angeht noch unschlüssig waren, sollten sie hierfür eine Konfigurationsmöglichkeit oder im einfachsten Fall eine entsprechende Konstante vorsehen. Für Test- und Entwicklungszwecke, können Sie erstmal von wenige Millisekunden ausgehen.

Erkennung von Krankheiten

Ihre zweite Aufgabe ist es, die Erkennung von Krankheiten der Hamster zu implementieren. Hierfür soll das System automatisch erkennen, wenn ein Hamster innerhalb eines gewissen Zeitraums alle Leckerli verweigert und in diesem Fall einen Tierarzt verständigen. Da die kooperierenden Tierärzte keine Webservice-Schnittstellen anbieten („Digitalisierung ist Neuland“), muss dieser Vorgang leider manuell erfolgen. Es genügt daher, wenn Sie Sie einen Eintrag in den Log des Servers einstellen, dass der betroffene Hamster wahrscheinlich erkrankt ist (stellen Sie aber sicher, dass der Name und der Besitzer des Hamsters im Logeintrag enthalten ist).

Wichtig: Für das Übungsblatt gehen wir schon bei einem endgültig verweigertem Fütterungsversuch (sprich: der Hamster hat drei mal nicht auf ein Leckerli reagiert) davon aus, dass ein Hamster krank ist.

Vorgehen

Für die Implementierung der transparenten Wiederholung und der Erkennung von Krankheiten von Hamstern lassen sich Bibliotheken für Fehlertoleranz sehr gut einsetzen. Eine jeweils geeignete Bibliothek ist in der Vorlage schon eingebunden.

Behandeln Sie einen Hamster bitte direkt als krank, sobald die Wiederholung der Fütterungsversuche fehlgeschlagen ist. Bitte verwenden Sie die von der Bibliothek angebotenen Funktionalitäten, suchen Sie sich jeweils eine passende aus und konfigurieren Sie diese. Für eine manuelle Implementierung bekommen Sie keine Punkte.

Tests

Sie finden im Ordner *tests* auch eine spezifische Testsuite *ResilienceRunnerHamster.jar*. Mit dieser können Sie die Resilienz Ihrer Lösung testen. Zu diesem Zweck arbeiten die Zufallszahl-

[1] Tatsächlich ist die Simulation bisher die einzige Implementierung, da die Entwicklung der Sensoren um festzustellen, ob der Hamster das Leckerli akzeptiert hat stecken geblieben ist.

Generatoren der Hamsterlib mit fest kodierten Seeds. Diese dürfen Sie daher nicht verändern, da ansonsten die Testsuite für die Resilienz selbst für eine korrekte Lösung fehlschlägt.

Allgemeine Hinweise

Das Testframework ruft Ihren Client aus Aufgabe Aufgabe 7.1 auf, um Fütterungsversuche zu unternehmen und zu prüfen, ob das Leckerli jetzt tatsächlich verfüttert worden ist. Die Aufgabe für die Erkennung der Krankheiten ist daher nur sinnvoll bearbeitbar, wenn Sie vorher schon Aufgabe Aufgabe 7.1 gelöst haben.

Hinweise C

Für dieses Übungsblatt steht kein Template für die Programmiersprache C zur Verfügung. Sie müssen dieses Übungsblatt daher in Java oder C# bestreiten.

Hinweise Java

Verwenden Sie für die Erkennung kranker Hamster das Framework Resilience4j, welches in der Vorlage bereits eingebunden ist. Die Vorlage enthält sogar schon etwas Code, um von dynamisch erstellten Instanzen von Circuit Breaker und Retry die Logs abzugreifen, und zwar genau so, dass das Testframework diese Logs lesen kann.

Hinweise C#

Verwenden Sie für die Erkennung kranker Hamster das Framework Polly, welches in der Vorlage bereits eingebunden ist.

[1] `https://github.com/jax-rs`

[2] `https://jersey.github.io/`