

Artemis 6.4.0

Course Overview



ge64baw

Courses > Praktikum: Grundlagen der Programmierung WS22/23 > Exercises > W11H03 - PinguChat

W11H03 - PinguChat

Hausaufgabe

Hard

Submission due:

7 months ago

Points: 5 of 6

Assessment: manual ?

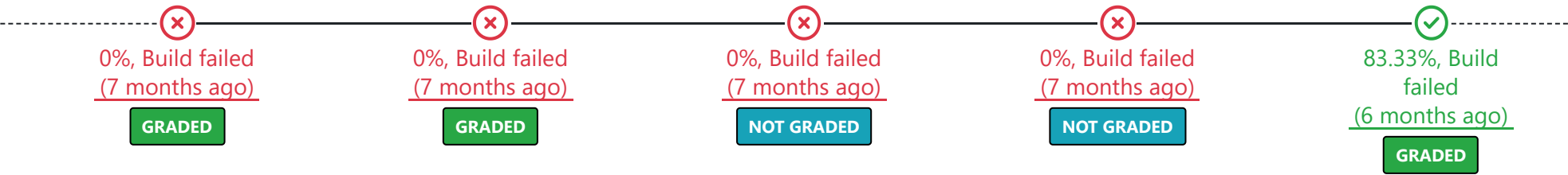
Complaint due:

6 months ago

 83.33%, Build failed (6 months ago)

GRADED

Recent results:



Show all results

PinguChat

Einige Pinguine haben sich entschieden, etwas Zeit bei uns an der Universität zu verbringen. Um mit ihren Freunden und Kollegen an der PUM weiter Kontakt halten zu können, möchten sie ein Chatprogramm entwickeln.

Der Ping-uin Jeff und der PinGUIn GUIseppe haben bereits den Server und die Benutzeroberfläche entwickelt. Da sie aber jetzt mit dem Umzug beschäftigt sind, brauchen sie deine Hilfe, um das Backend zu vervollständigen

Update 13.01.:

Um das Datum der in der Methode `getMessagesWithUser()` vom Server erhaltenen Messages in das geforderte `LocalDateTime`-Objekt umzuwandeln, sollte lediglich die Methode `LocalDateTime.parse()` auf den entsprechenden String vom Server angewandt werden.

Alle, die die Aufgabe noch am 13.01. begonnen haben, sollten in `DataHandler.login()` nachsehen, ob die in dieser Methode erstellte `HttpRequest` versucht, die URI "`http://carol.sse.cit.tum.de/api/users/me`" anzusprechen. Diese existiert nämlich nicht. Stattdessen sollte man "`http://carol.sse.cit.tum.de/api/user/me`" ansprechen. Ähnliches gilt für die Methode `DataHandler.register()` und "`http://carol.sse.cit.tum.de/api/users/register`" vs. "`http://carol.sse.cit.tum.de/api/user/register`"

Hinweise

- Eure Nachrichten sind unverschlüsselt gespeichert, passt also auf was ihr schreibt.
- Wenn Netzwerkoperationen Exceptions werfen, darf euer Programm abstürzen.
- Eine Version des Servers zur lokalen benutzung findet ihr [hier](#)
- Der Server ist erreichbar unter `carol.sse.cit.tum.de`
- Alle zu bearbeitenden Methoden sind mit TODO gekennzeichnet.
- Nützliche Links:
 - [Netcat](#) ist ein Linux Werkzeug zum Testen von Socket-Verbindungen.
 - [BurpSuite](#) ist ein Werkzeug zum Untersuchen von Verbindungen mit Websites.
 - <https://de.wikipedia.org/wiki/Byte-Reihenfolge>
- Die spitzen Klammern in den Beschreibungen der Bodies der Server-Antworten und das, was dazwischen steht, werden dann selbstverständlich durch den jeweils von Server berechneten String ersetzt.
- Damit ihr nicht ungewollt mit Nachrichten überflutet werdet, muss man sich erst auf Öffentlich setzen um mit anderen zu schreiben. Das geht, indem ihr euch [hier](#) anmeldet und an der Schnittstelle `/api/user/me/setpublic` über `Try Out` den `boolean` auf `true` setzt und anschließend `Execute` auswählt. Nun könnt ihr mit allen anderen öffentlichen Nutzern chatten. (Wenn der Link nicht funktioniert, liegt das wahrscheinlich daran, dass eue Browser automatisch einen HTTPS-Link daraus macht, obwohl ihr einen HTTP-Link braucht. Das sollte im Browser einstellbar sein, dass er das nicht mehr tut, sonst probiert andere Browser oder geht einfach direkt über die [IP-Adresse](#).)
- Das aktuelle Template findet ihr [hier](#)

API über HTTP:

`HTTP` ist ein häufig genutztes Protokoll zur Datenübertragung in einem Rechnernetz - also eine Reihe von Konventionen, wie Anfragen und Antworten der Kommunikation auszusehen haben u.Ä. Für diese Aufgabe müsst ihr `HTTP` nicht verstehen. Das Anfragen Erstellen und Empfangen ist bereits für euch übernommen worden. Hier werden die Java-eigenen Klassen/Interfaces `HttpRequest` - modelliert eine Anfrage an den Server - und `HttpResponse` modelliert die Antwort des Servers - verwendet. Ihr müsst dann lediglich mittels der beiden Methoden `HttpResponse.statusCode()` und `HttpResponse.body()` den Status und Inhalt der Antwort wie jeweils unten beschrieben weiterverarbeiten. Den Inhalt bekommt ihr als `String` im `JSON`-Format. Wenn ihr wollt, dürft ihr zu dessen Verarbeitung auch Libraries wie `org.json.*` importieren und verwenden. `JSONArray` und `JSONObject` aus diese Library sind für euch bereits in `DataHandler` importiert worden. Ihr dürft die `Strings` aber auch gerne per Hand bearbeiten. Das ist halt im Zweifelsfall etwas zeitaufwändiger und fehleranfälliger.

Ihr findet eine ausführlich Dokumentation der API mit Möglichkeiten zum Ausprobieren [hier](#) (wenn der Link nicht funktioniert, siehe oben). Hier findet ihr auch nochmal alle Informationen, wie die Header und Körper von Anfragen auszusehen haben. Das Wichtigste ist aber in den Beschreibungen der einzelnen Methoden unten für euch aufgelistet. Hier könnt ihr auch Anfragen ausprobieren und die Antworten sehen.

register

Die **register** Methode soll eine Anfrage auslösen, welche einen Nutzer für euch in der Datenbank des Servers anlegt. Falls ihr euer Passwort vergessen habt, kann diese Funktion auch genutzt werden, um euch ein neues Passwort zu erstellen. Das Passwort schickt der Server dann an eure TUM Email Adresse. Hierzu müsst ihr eine Anfrage an den `/api/users/register` Endpunkt stellen und den **register** übergebenen Nutzernamen und die TUM-Kennung mitschicken.

Das Schicken der Anfrage und Holen der Antwort ist bereits für euch implementiert worden. Die Antwort des Servers kommt in einer von zwei Formen:

▼ Wenn die Anfrage erfolgreich war:

Status-Code: 200

Body:

```
{
  "username": "<nutzername>",
  "id": <id>
}
```

▼ Wenn die Anfrage *nicht* erfolgreich war:

Status-Code: 422

Body:

```
{
  "detail": [
    {
      "loc": [
        "<ort des fehlers>",
        <eine zahl>
      ],
      "msg": "<fehler-nachricht>",
      "type": "<art des fehlers>"
    }
  ]
}
```

Alles, was ihr tun müsst, ist, genau dann **true** zurückzugeben, wenn die Anfrage erfolgreich war, sonst **false**.

requestToken

Um sich in Anfragen an die API authentifizieren zu können, braucht man einen Token. Diesen erhält man, wenn man seinen Nutzernamen und Passwort an den `/token` Endpunkt der API übergibt. Der Token, welchen man zurück erhält, ist jedoch nur für begrenzte Zeit gültig. Die **requestToken** Methode übernimmt das Anfragen dieses Tokens.

Das Schicken der Anfrage und Holen der Antwort ist bereits für euch implementiert worden. Die Antwort des Servers kommt in einer von zwei Formen:

▼ Wenn die Anfrage erfolgreich war:

Status-Code: 200

Body:

```
{
  "access_token": "<token>",
  "token_type": "<art des tokens>"
}
```

▼ Wenn die Anfrage *nicht* erfolgreich war:

Status-Code: 422

Body:

```
{
  "detail": [
    {
      "loc": [
        "<ort des fehlers>",
        <eine zahl>
      ],
      "msg": "<fehler-nachricht>",
      "type": "<art des fehlers>"
    }
  ]
}
```

Wenn die Anfrage erfolgreich war, sollt ihr nun den Token aus der Antwort auslesen und zurückgeben. Die Art des Tokens ist dabei irrelevant. Wenn die Anfrage erfolglos blieb, soll `null` zurückgegeben werden.

login

In der `login` Methode sollt ihr zuerst einen Token mit dem übergebenen Nutzernamen und Passwort anfragen. Falls diese Anfrage erfolgreich ist, werden im `DataHandler` die entsprechenden Klassenattribute gesetzt und die Nutzer ID vom `/api/user/me/` Endpunkt erfragt.

Das Schicken der Anfrage und Holen der Antwort ist bereits für euch implementiert worden. Die Antwort des Servers kommt in einer von zwei Formen:

▼ Wenn die Anfrage erfolgreich war:

Status-Code: 200

Body:

```
{
  "username": "<nutzername>",
  "id": <id>
}
```

▼ Wenn die Anfrage *nicht* erfolgreich war:

Status-Code: 422

Body:

```
{
  "detail": [
    {
      "loc": [
        "<ort des fehlers>",
        <eine zahl>
      ],
      "msg": "<fehler-nachricht>",
      "type": "<art des fehlers>"
    }
  ]
}
```

Wenn hier nun die Anfrage erfolgreich war, soll die ID in das entsprechende Attribut des `DataHandlers` geschrieben werden. Insgesamt soll die Methode genau dann `true` zurückgeben, wenn der gesamte Login erfolgreich war.

getContacts

Um die verfügbaren Kontakte zu erhalten, gibt es den Endpunkt `/api/users`. Dieser gibt alle auf öffentlich gestellten Nutzer zurück. Nutzt diesen Endpunkt, um eine `Map` mit den Nutzer IDs als Schlüssel und Nutzerobjekten zurückzugeben.

Das Schicken der Anfrage und Holen der Antwort ist bereits für euch implementiert worden. Die Antwort des Servers kommt in einer von zwei Formen:

▼ Wenn die Anfrage erfolgreich war:

Status-Code: 200

Body:

```
[
  {
    "username": "<nutzername 1>",
    "id": <id 1>
  },
  {
    "username": "<nutzername 2>",
    "id": <id 2>
  },
  ...
  {
    "username": "<nutzername n>",
    "id": <id n>
  }
]
```

▼ Wenn die Anfrage *nicht* erfolgreich war:

Status-Code: 422

Body:

```
{
  "detail": [
    {
      "loc": [
        "<ort des fehlers>",
        <eine zahl>
      ],
      "msg": "<fehler-nachricht>",
      "type": "<art des fehlers>"
    }
  ]
}
```

Aus dieser Antwort sollt ihr nun, wenn die Anfrage erfolgreich war, die entsprechende **Map** bauen.

getMessages

Die vergangenen Nachrichten einer Unterhaltung kann man mit dem Endpunkt `/api/messages/with/` laden. Dieser Endpunkt unterstützt auch das Unterteilen der Antwort. So kann man zum Beispiel nur 50 Nachrichten auf einmal laden. Ihr sollt hier die Nachrichten mit dem Nutzer mit der ID `id` als Liste zurückgeben beschränkt auf die in `count` übergebene Anzahl und die Seite `page`.

Das Schicken der Anfrage und Holen der Antwort ist bereits für euch implementiert worden. Die Antwort des Servers kommt in einer von zwei Formen:

▼ Wenn die Anfrage erfolgreich war:

Status-Code: 200

Body:

```
[
  {
    "from_id": <id des senders 1>,
    "to_id": <id des empfängers 1>,
    "text": "<inhalt der nachricht 1>",
    "id": <id der nachricht 1>,
    "time": "<datum der nachricht 1 (beispiel: 2023-01-13T10:39:10.900Z)>"
  },
  {
    "from_id": <id des senders 2>,
    "to_id": <id des empfängers 2>,
    "text": "<inhalt der nachricht 2>",
    "id": <id der nachricht 2>,
    "time": "<datum der nachricht 2>"
  },
  ...
  {
    "from_id": <id des senders n>,
    "to_id": <id des empfängers n>,
    "text": "<inhalt der nachricht n>",
    "id": <id der nachricht n>,
    "time": "<datum der nachricht n>"
  }
]
```

▼ Wenn die Anfrage *nicht* erfolgreich war:

Status-Code: 422

Body:

```
{
  "detail": [
    {
      "loc": [
        "<ort des fehlers>",
        <eine zahl>
      ],
      "msg": "<fehler-nachricht>",
      "type": "<art des fehlers>"
    }
  ]
}
```

Aus dieser Antwort sollt ihr nun, wenn die Anfrage erfolgreich war, die entsprechende **List** bauen.

Um das Datum in das geforderte **LocalDateTime**-Objekt umzuwandeln, sollt ihr die Methode `LocalDateTime.parse()` verwenden.

Socket

Um einen schnellen Austausch von Nachrichten zu ermöglichen, unterstützt unser Server auch eine direkte Socket Verbindung auf Port 1337. Zum Lesen und Schreiben von und zu dem Socket benutzen wir hier nicht wie in der Zentralübung `BufferedReader` und `PrintWriter`, sondern die beiden Klassen `DataInputStream` und `DataOutputStream` (siehe die Attribute `in` und `out` der Klasse `DataHandler`). `DataInputStream.read(byte[])` und `DataOutputStream.write(byte[])` übernehmen dabei jeweils ein `byte`-Array als Parameter. Erstere füllt das übergebene Array mit den vom Server erhaltenen Bytes, letztere schickt die im übergebenen Array enthaltenen Bytes an den Server. Im Gegensatz zur Zentralübung, in der wir Text verschickt haben, wollen wir also jetzt die genauen Bytes der Nachricht kontrollieren können. Dabei kodieren wir Nachrichten nach unserem Haus-eigenen Protokoll: dem PCP (Pingu Chat Protocol).

Die genaue Spezifikation findet ihr hier (Zahlen wie 0x2a sind genau ein Byte große = zwei Ziffern lange [Hexadezimalzahlen](#). Der Präfix "0x" zeigt dabei an, dass es sich hier um eine Hexadezimalzahl handelt, die restlichen Ziffern sind die Zahl selbst. [Umrechnung](#)):

▼ Protokoll

Legende:

S->C: Server schickt Nachricht an Client
 C->S: Client schickt Nachricht an Server

Nachrichtenaufbau:

Das erste Byte der Nachricht ist jeweils entweder 0x00, was für eine "Handshake-Nachricht" steht, oder 0x01, was für eine Text-Nachricht steht.

Handshake-Nachrichten sind zum kontrollieren der Verbindung, Text-Nachrichten schicken eine Message an einen Chat-Partner.

Handshake-Nachricht (0x00):

Das zweite Byte ist eine Zahl zwischen 0x00 und 0x05 oder 0xf0, 0xf1 oder 0xff.

Diese 9 Zahlen haben folgende Bedeutungen:

0x00: Server Hello

Die erste Nachricht, die man vom Server erhält, wenn man sich mit ihm verbindet.
 Erhält man diese Nachricht nicht zurück, weiß man,
 dass mit der Verbindung etwas schiefgelaufen ist.
 Als drittes und letztes Byte wird in diesem Fall die Versionsnummer des Servers geschickt.

Beispiel-Nachricht (S->C): 0x00 0x00 0x2a

Der Server begrüßt den Client mit seiner Versionsnummer 0x2a (= 42).

0x01: Client Hello

Diese Nachricht wird vom Server erwartet,
 sobald dieser sein Server-Hello an einen Client geschickt hat.
 Es wird kein weiteres Byte verschickt.

Beispiel-Nachricht (C->S): 0x00 0x01

Der Client begrüßt den Server bzw. bestätigt ihm den Erhalt von dessen Begrüßung.

0x02: Client Identification

Diese Nachricht sendet der Client an den Server, um sich bei ihm mit seiner ID anzumelden.
 Das dritte Byte dieser Nachricht soll eine Zahl k zwischen 1 und 8 sein,
 die die Anzahl an darauf noch folgenden Bytes ist.
 Darauf folgen dann k Bytes, die die gesamte ID des Clients enthalten.
 (Die ID wird dabei in Big-Endian-Kodierung versandt, also der uns aus Java bekannten.)

Beispiel-Nachricht (C->S): 0x00 0x02 0x02 0x13 0x37

Der Client identifiziert sich beim Server mit seiner ID 0x1337 (= 4919).
 Diese braucht 2 Bytes Platz, was im dritten Byte der Nachricht zu sehen ist.
 Man hätte die ID aber auch beispielsweise mit der Nachricht
 0x00 0x02 0x04 0x00 0x00 0x13 0x37 versenden können.

0x03: Client Authentication

Diese Nachricht sendet der Client an den Server, um sich mit dem zuvor erhaltenen Token
 zu authentifizieren. Das dritte und vierte Byte dieser Nachricht gibt die Länge k des
 Tokens an (also eine Zahl zwischen 0 und 65535), darauf folgen dann k Bytes mit dem Token.
 (Auch hier wird Big-Endian für die Kodierung verwendet.)

Beispiel-Nachricht (C->S): 0x00 0x03 0x00 0x01 0x45

Der Client authentifiziert sich beim Server mit einem Token von Länge 1B
 (siehe drittes und viertes Byte der Nachricht), nämlich dem Token 0x45.

0x04: Partner Switch

Der Client teilt dem Server mit, dass er gerne mit einem neuen Partner
 (i.d.R. anderer Client) Nachrichten austauschen würde.
 Das dritte Byte dieser Nachricht gibt die Länge 1 <= k <= 8 der ID des neuen Partners an,
 darauf folgen k Bytes ID.

Beispiel-Nachricht (C->S): 0x00 0x04 0x01 0x01

Der Client bittet den Server, ihn mit dem Client mit ID 0x01 zu verbinden.

0x05: Server Acknowledge

Diese Nachricht wird vom Server an den Client verschickt,
 wenn der Partnerwechsel stattfinden kann. Es werden hiernach keine weiteren Bytes verschickt.

Beispiel-Nachricht (S->C): 0x00 0x05

Der Server teilt dem Client mit, dass dieser nun mit seinem neuen Partner verbunden ist.

0xf0: Invalid Message (S->C)**0xf1: Authentication Failure (S->C)****0xff: Session End (S->C)**

Auf diese letzten drei folgen keine weiteren Bytes.

Text-Nachrichten (0x01):

Nachricht an/vom Chatpartner.

Das zweite und dritte Byte geben die Länge k der Nachricht (in Big-Endian) an, darauf folgen k Bytes mit der Nachricht in Byte-Form.

Beispiel-Nachricht (C->S): 0x01 0x00 0x05 0x48 0x61 0x6c 0x6c 0x6f
Der Client schickt die Nachricht "Hallo" (in UTF-8 Kodierung) an den Server, der diese dann an den aktuellen Gesprächspartner weiterleiten sollte.
Die UTF-8 Kodierung von "Hallo" ist 0x05 = 5 Bytes lang und lautet 0x48 0x61 0x6c 0x6c 0x6f.

Handshake Struktur:
Nach der ersten Verbindung mit dem Server (= dem Erstellen des Socket-Objektes auf Client-Seite) werden folgende Nachrichten in der angegebenen Reihenfolge ausgetauscht:


Server Hello
Client Hello
Client Identification
Client Authentication

Anschließend können Nachrichten wie Folgt verschickt werden:
(*: 0 oder mehr; +: 1 oder mehr; a|b: a oder b):
*{
 Partner Switch
 Server Acknowledgement
 *{
 S->C:Message|
 C->S:Message
 }
}

Hier eine Beispielverbindung

Exercise details

Release date:	Jan 13, 2023 18:30
Start date:	Jan 13, 2023 18:30
Submission due:	Jan 23, 2023 22:00
Assessment due:	Feb 15, 2023 18:00
Complaint due:	Feb 22, 2023 18:00

Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have **998** complaints left. 

How useful is this feedback to you?

