

AsyncTask, REST, HttpClient

Mag. DI Rainer Sickinger basierend auf den Unterlagen von
Mag. Dr. Bernhard Mayr, MBA

AsyncTasks Einführung

AsyncTask ist eine Möglichkeit, nebenläufige Prozesse in Android zu erzeugen, womit wir jedoch auch die Möglichkeit haben, nach Fertigstellung des Tasks, die Ergebnisse an die View weiterzureichen.

Probleme ohne Nebenläufigkeit

Der Aufruf von `Thread.sleep(5000)` simuliert einen länger andauernden Task. Wenn der simulierte Task fertig ist, wird ein Toast angezeigt.

```
public void startWithoutThread(View view) {
    try {
        Thread.sleep(5000);
        showToast("Thread Ready");
    } catch (InterruptedException e) {
        Log.d(TAG, e.getLocalizedMessage());
    }
}
```

Der Toast erscheint also erst 5 Sekunden, nachdem der Button geklickt wurde. Während dieser Zeit können jedoch auch keine anderen Buttons geklickt werden - das Userinterface ist eingefroren!

Gründe für das eingefrorene UI

- UI läuft im Main-Thread
- Jede Operation, die die App ausführt, läuft ebenfalls standardmäßig am Mainthread

Daraus folgt: Thread.sleep(5000) läuft auch im Mainthread und blockiert somit die UI.

- Dies ist problematisch, da das Android Betriebssystem erkennen kann, wenn ein UI nicht reagiert und die App schließen kann. Wir erhalten in diesem Fall einen ANR-Fehler (Application not responding).

Asynchrone Prozesse mit klassischem Threading

Ein Ansatz ist, sich der Java Boardmittel zu bedienen und die länger andauernde Tätigkeit in einen eigenen Thread auszulagern:

```
public void startSimpleThread(View view) {
    (new Thread() {
        @Override
        public void run() {
            TextView textView = findViewById(R.id.txt_msg);
            textView.setText("Thread Ready");
        }
    }).start();
}
```

Das UI ist in diesem Fall nicht mehr blockiert - während der neue Thread arbeitet, können die anderen Buttons der App problemlos gedrückt werden.

Das AsyncTask Framework

Die sicher eleganteste Variante ist die Verwendung des AsyncTask-Frameworks in Android. Dieses bietet über Callback-Methoden eine saubere Struktur!

Ein schematischer Ablauf

So könnte die Abarbeitung einer Hintergrundaufgabe mittels des Android AsyncTask-Frameworks aussehen:

- Zugriff auf das UI. Jetzt kann z.B. ein Fortschrittsbalken angezeigt werden.
- Auslagerung der länger andauernden Tätigkeit in einen eigenen Thread
- Immer wenn sich der aktuelle Fortschritt ändert, können wir auf den UI-Thread zugreifen und den Fortschritt setzen.
- Nach Beendigung des Hintergrundthreads erhalten wir das Ergebnis zurückgeliefert und können nun wieder das UI aktualisieren.

Man braucht grundsätzlich eine Klasse, welche von AsyncTask ableitet.

```
private class ServerTask extends AsyncTask<String, Integer, String> {  
    @Override  
    protected void onPreExecute() {  
        // here we could do some UI manipulation before the worker  
        // thread starts  
        super.onPreExecute();  
    }  
    @Override  
    protected void onProgressUpdate(Integer... values) {  
        // do some UI manipulation while progress is modified  
        super.onProgressUpdate(values);  
    }  
    @Override  
    protected String doInBackground(String... strings) {  
        // workhorse methode  
    }  
    @Override  
    protected void onPostExecute(String s) {  
        // called after doInBackground finishes  
    }  
}
```


Die Diamond-Parameter I

Bei der Ableitung von AsyncTask können wir unsere Klasse mit dem Diamond Operator typisieren: In unserem Fall <String, Integer, String>.

```
private class ServerTask extends AsyncTask<String, Integer, String> {
    @Override
    protected void onPreExecute() {
        // here we could do some UI manipulation before the worker
        // thread starts
        super.onPreExecute();
    }
    @Override
    protected void onProgressUpdate(Integer... values) {
        // do some UI manipulation while progress is modified
        super.onProgressUpdate(values);
    }
    @Override
    protected String doInBackground(String... strings) {
        // workhorse methode
    }
    @Override
    protected void onPostExecute(String s) {
        // called after doInBackground finishes
    }
}
```

Die Diamond-Parameter II

- Der erste Parametertyp (String) ist der Eingangstyp der Methode `doInBackground`.
- Der zweite Typ (Integer) ist der Eingangstyp der Methode `onProgressUpdate`.
- Der dritte Typ (String) ist der Rückgabertyp der Methode `doInBackground` und der Eingangstyp der Methode `onPostExecute`.

Hier kann man auch das Muster erkennen: Der Rückgabewert der Methode `doInBackground` kommt als Eingangswert in den Aufruf der Methode `onPostExecute`. D.h. innerhalb von `onPostExecute` kann einerseits auf die Ergebnisse des Hintergrundthreads und andererseits auch auf die UI-Komponenten zugegriffen werden.

Die Diamond-Parameter III

BEACHTEN: Bei der Typisierung müssen immer Objektdatentypen verwendet werden (Integer, Boolean, String, etc.) für primitive Datentypen sind die entsprechenden Wrapperklassen zu verwenden
- Void kann auch verwendet werden.

Die einzelnen Methoden der Klasse AsyncTask I

- **onPreExecute:** Wird beim Start des AsyncTask Objekts aufgerufen. Hier kann auf das UI zugegriffen werden und z.B. eine Fortschrittsbalken eingeblendet werden.
- **doInBackground:** Hier erfolgt die länger andauernde Arbeit im Hintergrundprozess. Die Methode liefert die Ergebnisse mittels `return` an die im Anschluss aufgerufene Methode `onPostExecute` zurück. Diese Methode kann z.B. für den Download oder die Abfrage an einen Webservice verwendet werden.

Die einzelnen Methoden der Klasse AsyncTask II

- `onPostExecute`: Wird nach Abschluss der nebenläufigen Arbeit aufgerufen. Sie erhält beim Aufruf die Ergebnisse von `doInBackground` und kann das UI manipulieren (z.B. die heruntergeladenen Ergebnisse anzeigen.)
- `onProgressUpdate`: läuft auf dem UI-Thread und wird aufgerufen, wenn innerhalb der Methode `doInBackground` die Methode `publishProgress` aufgerufen wird.

AsyncTask starten

- Der AsyncTask kann in einer eigenen Klasse definiert werden
- oder also innere Klasse in einer Activity.

Wird der Task nur in **einer** Activity benötigt → innere Klasse.

Wird der Task in verschiedenen Activitys benötigt → eigene Klasse.

Den AsyncTask kann man wie folgt starten:

```
public void startAsyncTask(View view) {
    ServerTask task = new ServerTask();
    task.execute("AsyncTask Thread ready");
}
```

Beispiel: AsyncTask mit Fortschrittsbalken

Im Layout müssen wir eine View Komponente hinzufügen. Wir verwenden einen horizontalen Fortschrittsbalken - wir könnten jedoch auch jede andere View Komponente dafür verwenden und zB. den Fortschritt in Prozent in einer TextView anzeigen.

```
<ProgressBar
    android:id="@+id/progressBar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1" />
```

Taskstruktur

Im Konstruktor des Tasks geben wir die Activity mit um Zugriff auf die Steuerelemente zu bekommen.

```
public class ProgressTask extends AsyncTask<String, Integer, String> {
    private MainActivity activity ;
    private ProgressBar mProgressBar;

    public ProgressTask(MainActivity activity ) {
        this.activity = activity ;
    }
    @Override
    protected void onPreExecute() {
        super.onPreExecute();
    }
    @Override
    protected void onProgressUpdate(Integer ... values) {
        super.onProgressUpdate(values);
    }
    @Override
    protected String doInBackground(String... strings) {
    }
    @Override
    protected void onPostExecute(String s) {
        super.onPostExecute(s);
    }
}
```


onPreExecute

In der Methode `onPreExecute` initialisieren wir alle unsere Steuerelemente.

```
@Override
protected void onPreExecute() {
    mProgressBar =
        activity.findViewById(R.id.progressBar);
    mProgressBar.setVisibility(View.VISIBLE);
    super.onPreExecute();
}
```

onProgressUpdate

In der Methode `onProgressUpdate` muss der Fortschrittsbalken aktualisiert werden.

```
@Override
protected void onProgressUpdate(Integer... values) {
    mProgressBar.setProgress(values[0]);
    super.onProgressUpdate(values);
}
```

doInBackground

In der Methode `doInBackground` muss die Methode `publishProgress` aufgerufen werden, immer wenn sich der Fortschritt ändert. Android ruft dann automatisch `onProgressUpdate` auf.

```
@Override
protected String doInBackground(String... strings) {
    for (int i=0; i<10; i++) {
        publishProgress(i*10);
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Log.d("progress", e.getMessage());
        }
    }
    String msg = strings[0];
    return msg;
}
```

onPostExecute

Nachdem die Arbeit im Hintergrund erledigt ist, sollte der Fortschrittsbalken natürlich auch wieder verschwinden. Dazu müssen wir die Methode `onPostExecute` anpassen.

```
@Override
```

```
protected void onPostExecute(String s) {  
    mProgressBar.setVisibility(View.INVISIBLE);  
    super.onPostExecute(s);  
}
```

Task starten



Gestartet werden kann der Task dann zum Beispiel wie folgt:

@Override

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    ProgressTask pt = new ProgressTask(this);  
    pt.execute("test");  
}
```

AsyncTask ist deprecated

Android Developers > Docs > Reference

War das hilfreich?  


AsyncTask

Added in API level 3
Deprecated in API level 30

[Kotlin](#) | [Java](#)

```
public abstract class AsyncTask  
extends Object
```

[java.lang.Object](#)
↳ [android.os.AsyncTask](#)<Params, Progress, Result>

 This class was deprecated in API level 30.
Use the standard [java.util.concurrent](#) or [Kotlin concurrency utilities](#) instead.

AsyncTask was intended to enable proper and easy use of the UI thread. However, the most common use case was for integrating into UI, and that would cause Context leaks, missed callbacks, or crashes on configuration changes. It also has inconsistent behavior on different versions of the platform: swallows exceptions from `doInBackground`, and does

Google markiert die Android AsyncTask-API in Android 11 als veraltet und schlägt vor, stattdessen `java.util.concurrent` zu verwenden.

Gründe warum AsyncTask als deprecated markiert wurde

<https://www.techyourchance.com/asynctask-deprecated/>

Die Lösung!

```

public class TaskRunner {
    private final Executor executor = Executors.newSingleThreadExecutor(); // change according to your
                                   requirements
    private final Handler handler = new Handler(Looper.getMainLooper());

    public interface Callback<R> {
        void onComplete(R result);
    }

    public <R> void executeAsync(Callable<R> callable, Callback<R> callback) {
        executor.execute(() -> {
            final R result = callable.call();
            handler.post(() -> {
                callback.onComplete(result);
            });
        });
    }
}

```

Die Lösung!

```
class LongRunningTask implements Callable<MyPojo> {  
    private final String input;  
  
    public LongRunningTask(String input) {  
        this.input = input;  
    }  
  
    @Override  
    public MyPojo call() {  
        // Some long running task  
        return myPojo;  
    }  
}
```

Die Lösung!

```
// in ViewModel
taskRunner.executeAsync(new LongRunningTask(input), (data) -> {
    // MyActivity activity = activityReference.get();
    // activity.progressBar.setVisibility(View.GONE);
    // populateData(activity, data);

    loadingLiveData.setValue(false);
    dataLiveData.setValue(data);
});
```

Die Lösung!

```
// in Activity
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main_activity);

    viewModel = ViewModelProviders.of(this).get(MyViewModel.class);
    viewModel.loadingLiveData.observe(this, (loading) -> {
        if (loading) {
            progressBar.setVisibility(View.VISIBLE);
        } else {
            progressBar.setVisibility(View.GONE);
        }
    });

    viewModel.dataLiveData.observe(this, (data) -> {
        populateData(data);
    });
}
```

Eine andere Lösung

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Handler handler = new Handler(Looper.getMainLooper());

executor.execute(new Runnable() {
    @Override
    public void run() {

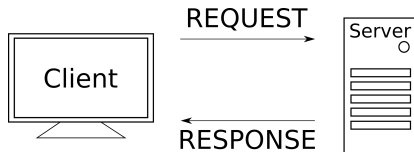
        //Background work here

        handler.post(new Runnable() {
            @Override
            public void run() {
                //UI Thread work here
            }
        });
    }
});
```

Eine weitere Lösung

```
new Thread(new Runnable() {  
    @Override  
    public void run() {  
        // do your stuff  
        runOnUiThread(new Runnable() {  
            public void run() {  
                // do onPostExecute stuff  
            }  
        });  
    }  
}).start();
```

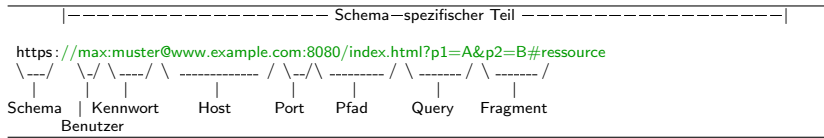
- HTTP steht für Hyper Text Transfer Protocol.
- HTTP ist ein **zustandsloses** Datenübertragungsprotokoll.
- HTTP ist im ISO/OSI-Schichtmodell im Layer 7 (Anwendungslayer) angesiedelt.
- Wird unter anderem eingesetzt um Hypertext-Dokumente (Websites) aus dem WWW in einen Webbrowser zu laden.



Um einen validen HTTP-Request absetzen zu können muss dieser folgende Dinge beinhalten

- URL (Uniform Resource Locator)
- METHOD
- HEADER
- BODY

Mit einer URL identifiziert der Client die Dinge mit welchen er interagieren will. Diese Dinge nennt man **Ressourcen**.



Mit der **METHOD** gibt der Client an welche Aktion der Server ausführen soll. Folgende **Methoden** werden häufig benötigt:

- GET - Der Server wird aufgefordert eine Ressource zu liefern.
- POST - Der Server wird aufgefordert eine neue Ressource zu erstellen.
- PUT - Der Server wird aufgefordert eine Ressource zu verändern.
- DELETE - Der Server wird aufgefordert eine Ressource zu löschen.

Im HTTP-Header werden zusätzliche Informationen über den HTTP-Request übergeben. Es können **unter anderem** folgende Informationen übergeben werden.

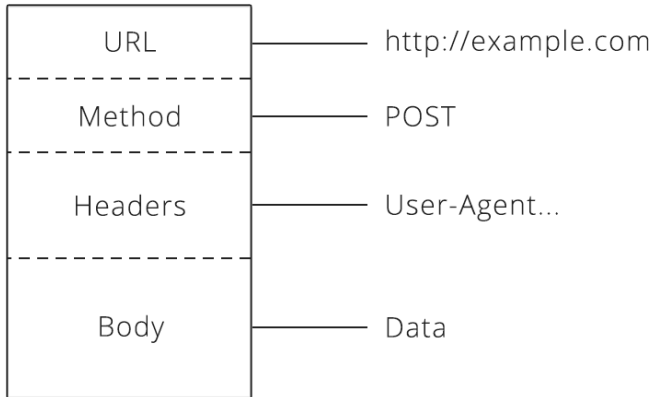
- User-Agent - Informationen über den Client (OS, Browser, ...)
- Accept - Gibt zum Beispiel an, welche Inhaltstypen der Client verarbeiten kann.
- Accept-Charset - Gibt an, welches Charset der Client verarbeiten kann.
- usw...

Weitere Informationen:

https://de.wikipedia.org/wiki/Liste_der_HTTP-Headerfelder

Der **HTTP-Body** beinhaltet die Daten, welche bei einer HTTP-Transaktion übermittelt werden sollen.

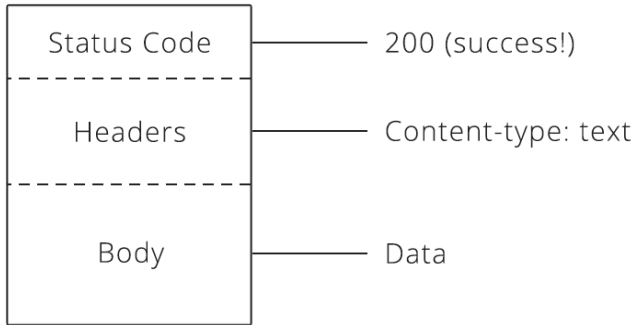
Aufbau Http-Request



Request

```
POST /genapp/customers/  
Host: www.example.com  
Content-Type: application/json  
Content-Length: 127 1
```

```
{  
  "customers":  
  {  
    "firstName": "Joe",  
    "lastName": "Bloggs",  
    "fullAddress":  
    {  
      "streetAddress": "21 2nd Street",  
      "city": "New York",  
      "state": "NY",  
      "postalCode": 10021  
    }  
  }  
}
```



Response

Weitere Informationen zu den Headerfeldern:

https://de.wikipedia.org/wiki/Liste_der_HTTP-Headerfelder

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

REST

- REST steht für Representation State Transfer
- REST ist ein Architekturstil für verteilte Hypermedia-Systeme

Mit REST kann man verteilte Systeme entwickeln, welche einfach zu verstehen und zu skalieren sind.

- Clients können sich darauf verlassen wie sich ein Service verhält.
- Die Kommunikation ist einfach und lesbar.
- Skalierung geht mehr in die Richtung „Einfach neuen Service hinzufügen“ und weniger in Richtung „Wir müssen die Architektur verändern“.

REST sollte sich bekannt anfühlen

- REST ist ein Architekturstil und kein Standard.
- REST war die Grundidee von HTTP
- Es werden altbekannte Konzepte verwendet
 - URIs, URLs
 - Hypertext
 - Content Types
 - HTTP Codes
 - HTTP Methoden
- HTTP kann auch in einem nicht REST-konformen Stil verwendet werden. Beispielsweise wird HTTP bei SOAP (Simple Object Access Protocol) einfach zum Übertragen von Daten verwendet. HTTP hat aber jede Menge an tollen Features welche oft ignoriert werden.

REST besteht aus fünf fundamentalen Konzepten



Ressourcen



Hypertext



Operationen

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Representation



Statuslos

Ressourcen

Jedes „Ding“ in REST nennt man **Ressource**. Jede Ressource bekommt eine eindeutige Kennung:

- /game/robots/four-hand-joe
- /game/spaceships/destroyers/98

URI und URL

Eine URL ist ein Konzept für Kennungen im WWW.

- Eine **URI** identifiziert eine Ressource über die Location und den Namen.
- Eine **URL** ist eine URI welche festlegt wo sich eine Ressource befindet, wie sie heißt und wie sie abzurufen ist.

Beispiel für eine URL:

`http://mygame.com/game/robots/four-hand-joe`

Diese URL gibt an, wo (`mygame.com/game/robots`) sich die Ressource `four-hand-joe` befindet und wie man diese abrufen kann (HTTP).

REST sagt: Alles was der Client abrufen können soll muss eine Kennung erhalten.

- Auch eine Menge von Dingen verdient eine Kennung:
 - /game/robots
 - /game/spaceships?visible=false
- Ressourcen können statisch sein oder sich über die Zeit verändern:
 - /game/robots/four-hand-joe
 - /game/spaceships/upgrades/2312
- Eine Ressource kann über verschiedene URLs referenziert werden:
 - /game/robots/four-hand-joe
 - /game/spaceships/4/crew/four-hand-joe

Representation einer Ressource

- Eine Ressource muss irgendwie representiert werden. Diese Representation kann zum Beispiel über XML, JSON oder HTML erfolgen.
- Der Client gibt bei einem HTTP-Request über den Header bekannt wie die Ressource representiert werden soll. Kann der Server die Representation nicht liefern wird ein Fehler zurückgegeben.

```
GET /game/robots HTTP/1.1
```

```
HOST mygame.com
```

```
Accept: application/json
```

- Der Typ/Form der Representation muss als MIME type spezifiziert werden. ¹

¹<https://www.freeformatter.com/mime-types-list.html> ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ↺ 🔍 ↻

REST sagt: Eine Ressource kann verschieden representiert werden:

GET /game/robots HTTP/1.1

HOST mygame.com

Accept: application/json

GET /game/robots HTTP/1.1

HOST mygame.com

Accept: application/xml

GET /game/robots HTTP/1.1

HOST mygame.com

Accept: text/html

Wird ein Typ nicht unterstützt muss der Server 406 Not Acceptable zurückgeben.

REST definiert vier Standardoperationen um mit Ressourcen zu interagieren:

- GET: Eine Representation einer Ressource abrufen.
- POST: Eine neue Ressource erstellen.
- PUT: Eine Ressource verändern.
- DELETE: Eine Ressource löschen.

Diese Operationen werden auch im HTTP-Standard festgelegt. Dort heißen diese jedoch **verbs**.

Beispiele:

- GET /game/robots/four-hand-george
- POST an /robots um einen neuen Roboter anzulegen.

Jede Ressource unterstützt eine oder mehrere Operationen.

Bei REST muss das Informationssystem so implementiert werden, dass man mit den Operationen (GET, POST, ...) und einer Menge von Ressourcen alles abbildet.

Beispiel: Wir wollen das Raumschiff mit der ID `ship-1` zum Mars schicken.

- POST nach `/game/spaceships/ship-1/flightpath` mit `destination=mars` im body legt einen neuen Flugpfad an.
- Nun kann man mit einem GET auf `/game/spaceships/ship-1/flightpath` den Flugpfad abfragen. Mit diesem GET erfahren wir auch, dass der aktuelle Flugpfad zum Mars die ID 1234 hat.
- Mit einem DELETE auf `/game/spaceships/ship-1/flightpath/1234` können wir den Flug abbrechen.

Sichere und Idempotente Operationen

REST definiert einige Regeln bezüglich der Standardoperationen (GET, POST, ...).

Eine Operation kann entweder **sicher** und/oder **idempotent** sein.

- **Sicher**: Eine Operation ist **sicher**, wenn sie keine Ressource verändert.
- **Idempotent**: Eine Operation ist idempotent, wenn sie bei mehrfacher Hintereinanderausführung das gleiche Ergebnis wie bei einer einzigen Ausführung liefert.

Beispiel: Wenn $a = 1$ dann ist die Multiplikation eine idempotente Operation da: $a \cdot a = a^n$ mit $n \in \mathbb{N}$

Der Client kann folgendes erwarten bezüglich der Operationen:

- GET: **sicher und idempotent**. GET modifiziert nichts und es passiert nichts, wenn man GET öfters aufruft.
- DELETE: **nicht sicher aber idempotent**. DELETE modifiziert Ressourcen es hat aber keinen zusätzlichen Effekt, wenn DELETE öfter als einmal aufgerufen wird.
 - DELETE /game/spaceships/ship-1/flightpath/1234 löscht den Flugpfad, wenn die Aktion zum erstem mal ausgeführt wird.
 - weitere Aufrufe werden von dem Server ignoriert.
- PUT: **nicht sicher aber idempotent** PUT modifiziert Ressourcen es hat aber keinen zusätzlichen Effekt, wenn PUT öfter als einmal aufgerufen wird.
- POST: **nicht sicher und nicht idempotent** POST modifiziert Ressourcen und hat bei mehrfacher Ausführung auch weitere Auswirkungen auf das System.

Applikationen, welche REST verwenden, können Links verfolgen, um mehr Informationen zu erhalten:

```
{
  "employeeId": 1,
  "firstName": "Lokesh",
  "lastName": "Gupta",
  "email": "howtodoinjava@gmail.com",
  "_links": {
    "self": {
      "href": "http://localhost:8080/1"
    },
    "report": {
      "href": "http://localhost:8080/employees/1/report"
    }
  }
}
```

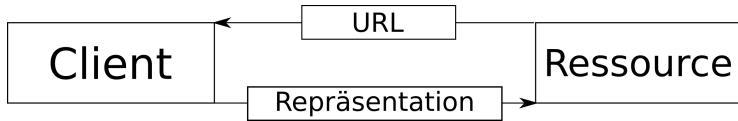
Statuslos

REST sagt: Der Server hält keinerlei Status der Applikation, sondern nur den Status der Ressourcen.

- Nur der Client, nicht der Server, hält Zustandsinformationen.
- Jede REST-Nachricht enthält alle Informationen, die für den Server bzw. Client notwendig sind, um die Nachrichten zu verstehen.
- Jede Anfrage an den Server ist in sich geschlossen.
- Begünstigt Skalierbarkeit des Services
 - Eingehende Anfragen können in Bezug auf die Lastenverteilung einfach auf beliebige Maschinen verteilt werden.
 - Weiterhin begünstigt wird die Ausfallsicherheit, weil die Zustandslosigkeit fordert, dass transaktionale Datenübertragung in einem einzigen Seitenaufruf erfolgt.

Durch diese Zustandslosigkeit erhält man Skalierbarkeit!

WARUM heißt REST so wie es heißt?



- Der Client referenziert eine Ressource mithilfe einer URL.
- Eine Repräsentation der Ressource wird zurückgegeben.
- Die Repräsentation bringt den Client in einen Zustand.
- Der Client fordert eine weitere Ressource via eines Hyperlinks an.
- Die neue Repräsentation der angeforderten Ressource bringt den Client in einen neuen Zustand.
- Die Clientapplikation ändert (transformiert) den Status mit jeder Ressourcenrepräsentation.
- **Repräsentation State Transfer**

HTTPClient

- Kaum eine App kommt ohne den Zugriff auf weitere Webservices aus.
- Ein zentrales Element der mobilen Softwareentwicklung ist daher der Zugriff auf Webservices.
- Android stellt die Klasse `HttpURLConnection` zur Verfügung um HTTP Requests absetzen zu können.
- Wenn ein Request abgesetzt wird, wird ein `InputStream` vom Android SDK zurückgegeben.

Permission

Da wir nun auf das Internet zugreifen wollen, müssen wir zu allererst unserer App die entsprechende Berechtigung verschaffen. Dazu genügt ein Eintrag in der Manifest-Datei, da die Permission Internet zu den Standard-Permissions und nicht zu den gefährlichen Permissions zählt. Eine zusätzliche Abfrage der Permission im Code ist daher nicht erforderlich.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.eaustria.http.demo" >
    <uses-permission android:name="android.permission.INTERNET" />
    <application>
        ...
    </application>
</manifest>
```

Änderungen ab Android API Level 28

- Seit Android 9 (API Level 28) ist der Zugriff auf nicht verschlüsselte Webseite standardmäßig nicht mehr erlaubt.
- Um den Zugriff auf http ohne SSL-Verschlüsselung zu erlauben, muss man im Manifest einen entsprechenden Eintrag hinzufügen: `android:usesCleartextTraffic="true"`

```
<uses-permission android:name="android.permission.INTERNET" />
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportRtl="true"
    android:theme="@style/AppTheme"
    android:usesCleartextTraffic="true">
    <activity android:name=".MainActivity">
        ...
    </activity>
</application>
```

Ablauf für den Abruf eines Webserices in Android

- **Eigenen Thread starten:** In aktuellen Androidversionen müssen sämtliche „Internet-Tätigkeiten“ in einem eigenen Thread ausgeführt werden, da diese länger dauern und somit das UI blockieren könnten.
- Connection auf Webservice aufbauen
- Responsecode prüfen
- InputStream auswerten
- Erhaltene Daten verarbeiten

Aufgrund der Forderung von Android nach Nebenläufigkeit beim Zugriff auf Webservices bietet sich der Einsatz vom AsyncTask-Framework an.

Beispiel GET-Request I

Beim GET-Request werden die Parameter des Aufrufs direkt an die URL angehängt! Wir wollen uns alle Repositories eines bestimmten Users auf github anzeigen lassen.

Beispiel GET-Request II

Dafür verwenden wir die API von Github ² und führen einen GET-Request auf folgende URL durch:

`https://api.github.com/users/[USERNAME]/repos.`

Anstelle von [USERNAME] fügen wir einen Benutzernamen (zB. unseren eigenen) als Aufrufparameter ein. Wir können aber die öffentlichen Repositories von jedem beliebigen Benutzer abfragen.

²<https://developer.github.com/v3/repos/#list-user-repositories>

Beispiel GET-Request III

- The Klasse `URLConnection` führt die konkrete Abfrage durch. Die Anfrageparameter werden als Teil der URL mitübergeben.
- Sobald die Methode `getResponseCode` aufgerufen wird, wird die Abfrage durchgeführt.
- Der Responsecode beinhaltet die Antwort vom Server. Dieser Code entspricht den HTML Codes, die auch als Konstante zur Verfügung stehen.
- Als Antwort erhält man einen `InputStream`, von dem die Daten gelesen werden können.
- Viele Webservices bieten die Antwort im JSON Format an, weshalb in der Regel danach das Parsen der JSON Antwort erforderlich ist (im Demobeispiel erfolgt dies in der Methode `outputRepoNames`).

Beispiel GET-Request IV

```

@Override
protected String doInBackground(String... strings) {
    Log.d(TAG, "entered doInBackground");
    String username = strings[0];
    Log.d(TAG, "url: " + URL + username + "/repos");
    String sJson = "";
    try {
        HttpURLConnection connection =
            (HttpURLConnection) new URL(URL + username + "/repos").openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Content-Type", "application/json");
        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(connection.getInputStream()));
            sJson = readResponseStream(reader);
        }
    } catch (IOException e) {
        Log.d(TAG, e.getMessage());
    }
    return sJson;
}

```

Beispiel GET-Request V

```
private String readResponseStream(BufferedReader reader) throws IOException {
    Log.d(TAG, "entered readResponseStream");
    StringBuilder stringBuilder = new StringBuilder();
    String line = "";
    while ( ( line = reader.readLine() ) != null ) {
        stringBuilder.append(line);
    }
    return stringBuilder.toString();
}
```

Beispiel GET-Request VI

```
@Override
protected void onPostExecute(String s) {
    Log.d(TAG, "entered onPostExecute");
    outputRepoNames(s, mRepoList);
    super.onPostExecute(s);
}
```

Beispiel GET-Request VII

```
private void outputRepoNames(String s, EditText mRepoList) {  
    Log.d(TAG, "entered outputRepoNames");  
    Log.d(TAG, s);  
    StringBuilder stringBuilder = new StringBuilder();  
    try {  
        JSONArray array = new JSONArray(s);  
        for (int i=0 ; i < array.length() ; i++) {  
            JSONObject jsonObject = array.getJSONObject(i);  
            stringBuilder.append(jsonObject.optString("full_name"));  
            stringBuilder.append("\n\n");  
        }  
    } catch (JSONException e) {  
        Log.d(TAG, e.getMessage());  
    }  
    mRepoList.setText( stringBuilder.toString());  
}
```

Gesamtes Beispiel ist verfügbar unter
<http://sickinger-solutions.at/schule/exampleGET.java>.

POST Aufruf I

Der POST-Aufruf erfolgt ähnlich wie GET. Der einzige Unterschied besteht darin, dass ein Body mit Daten angefügt werden kann. Folgender Code zeigt, wie man einen Body aus einem JSON erzeugen kann.

```
private byte[] getPostBytes() throws JSONException {  
    JSONObject postParams = new JSONObject();  
    postParams.put("paramA", "first param value");  
    postParams.put("paramB", 123456);  
    String body = postParams.toString();  
    Log.d(TAG, "body: " + body);  
    return body.getBytes();  
}
```

POST Aufruf II

Folgender Aufruf zeigt die Abgabe des POST requests.

```
private byte[] getPostBytes() throws JSONException {  
private HttpURLConnection getPostConnection( byte[] data, String url)  
    throws IOException {  
  
    HttpURLConnection connection =  
        (HttpURLConnection) new URL(url).openConnection();  
    connection.setDoOutput(true);  
    connection.setRequestMethod("POST");  
    connection.setRequestProperty("Content-Type", "application/json");  
    connection.setFixedLengthStreamingMode(data.length);  
    connection.getOutputStream().write(data);  
    connection.getOutputStream().flush();  
    return connection;  
}
```

POST Aufruf II

PUT und DELETE Aufrufe erfolgen genauso wie GET oder POST,
nur dass die entsprechende Request Methode gesetzt werden muss:
`connection.setRequestMethod("PUT")`

oder

`connection.setRequestMethod("DELETE")`