

RABBITMQ IS MORE THAN
A SIDEKIQ REPLACEMENT



Stanko Krtalic Rusendic

 github.com/stankek

 [@monorkin](https://twitter.com/monorkin)

 hey@stanko.io

 stanko.io

Before we continue, a quick introduction.

My name is Stanko. I am a software engineer at FloatingPoint.

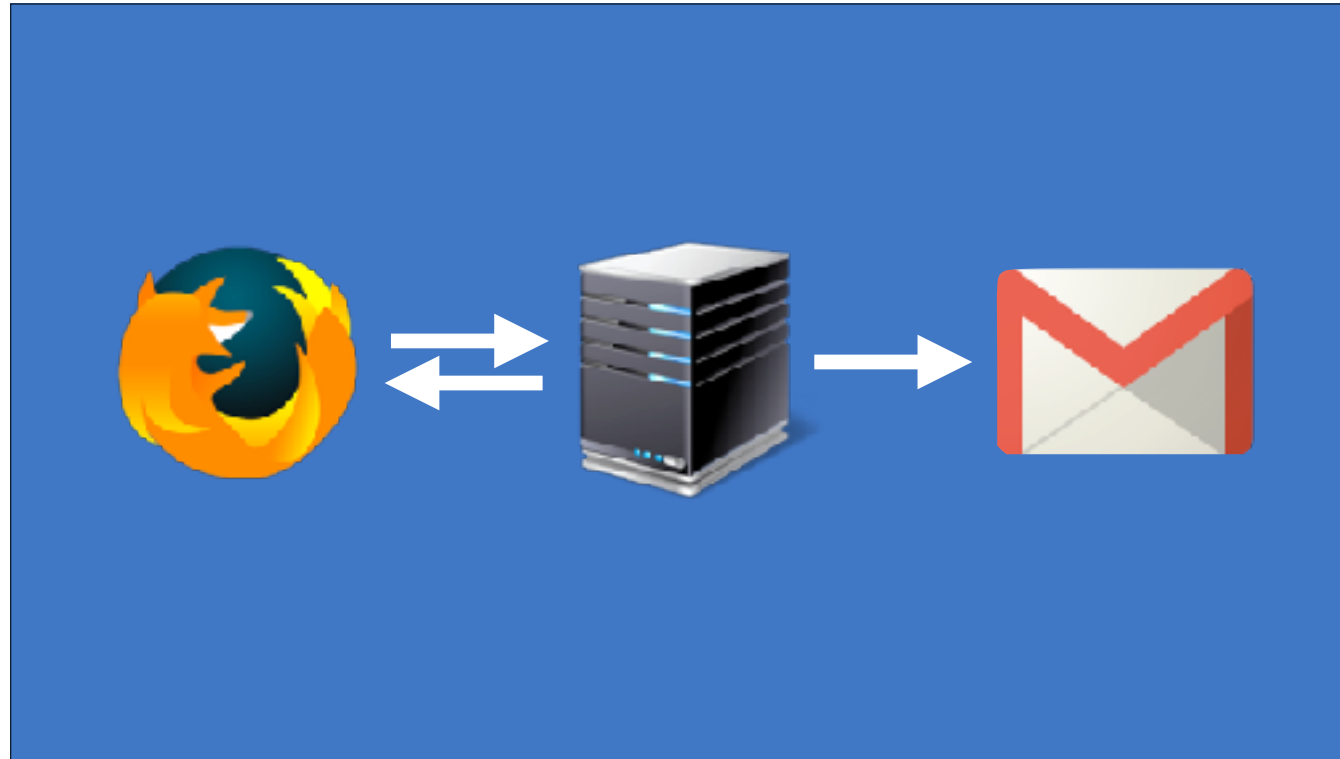
I mostly work with Ruby, Elixir and Rust.



Ok, so why do we need Sidekiq or RabbitMQ? Well let's start from what they have in common. You can use them for background workers.

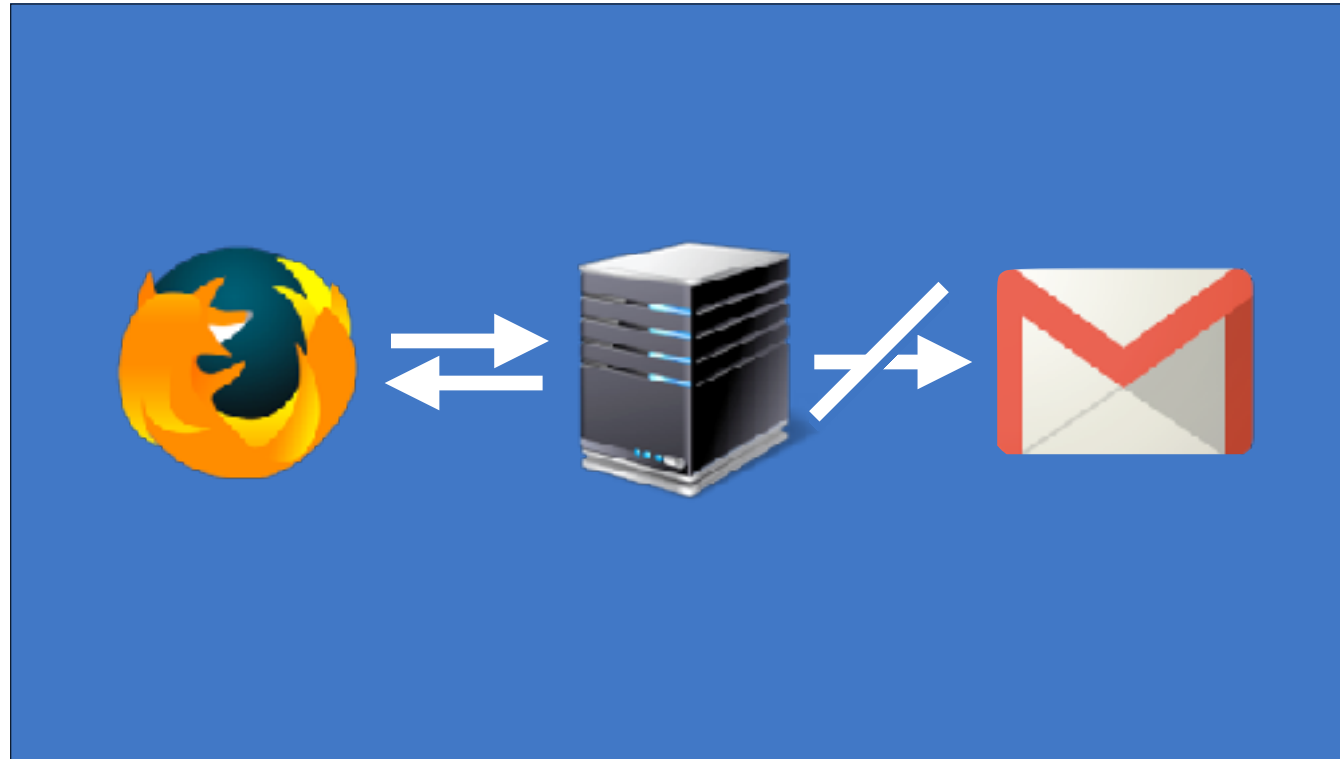
Asynchronous execution

Background workers enable you to execute code asynchronously and in parallel.

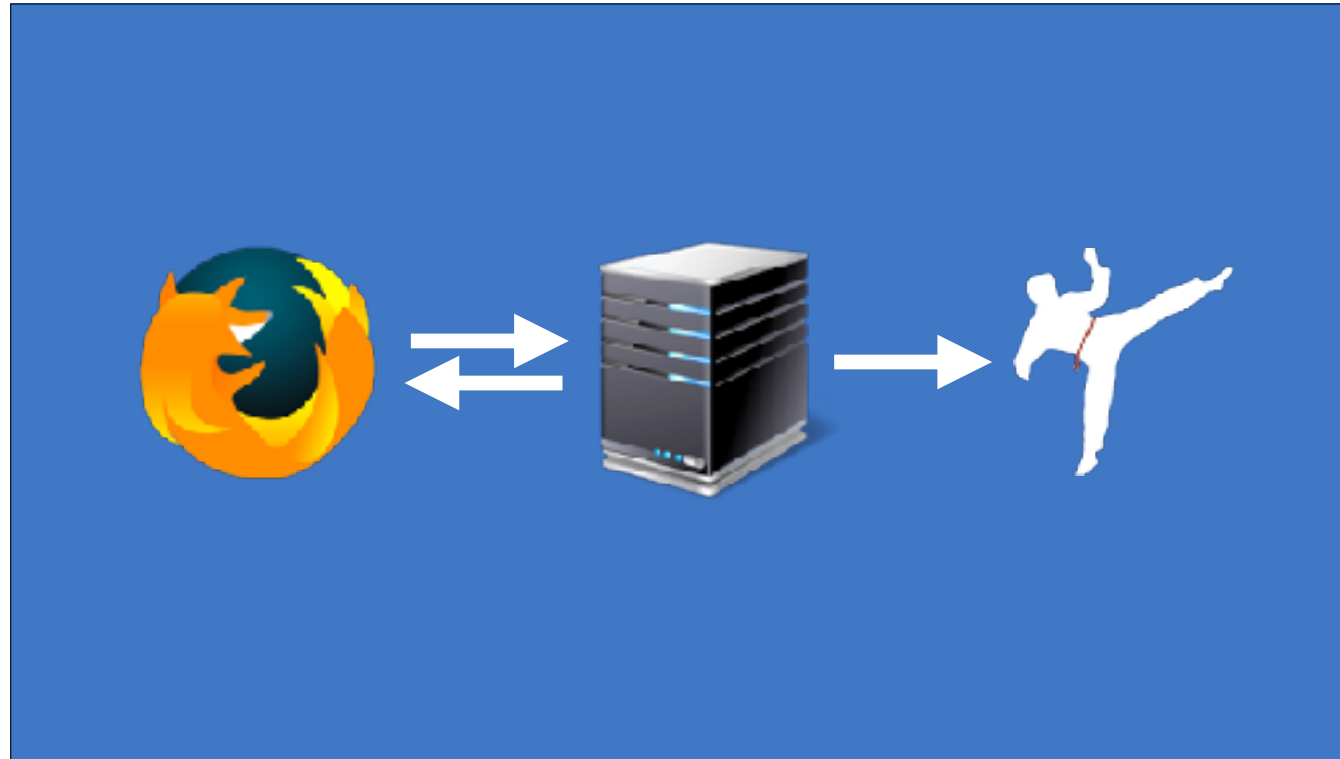


A typical example for using background workers is sending out mail, e.g. a signup confirmation.

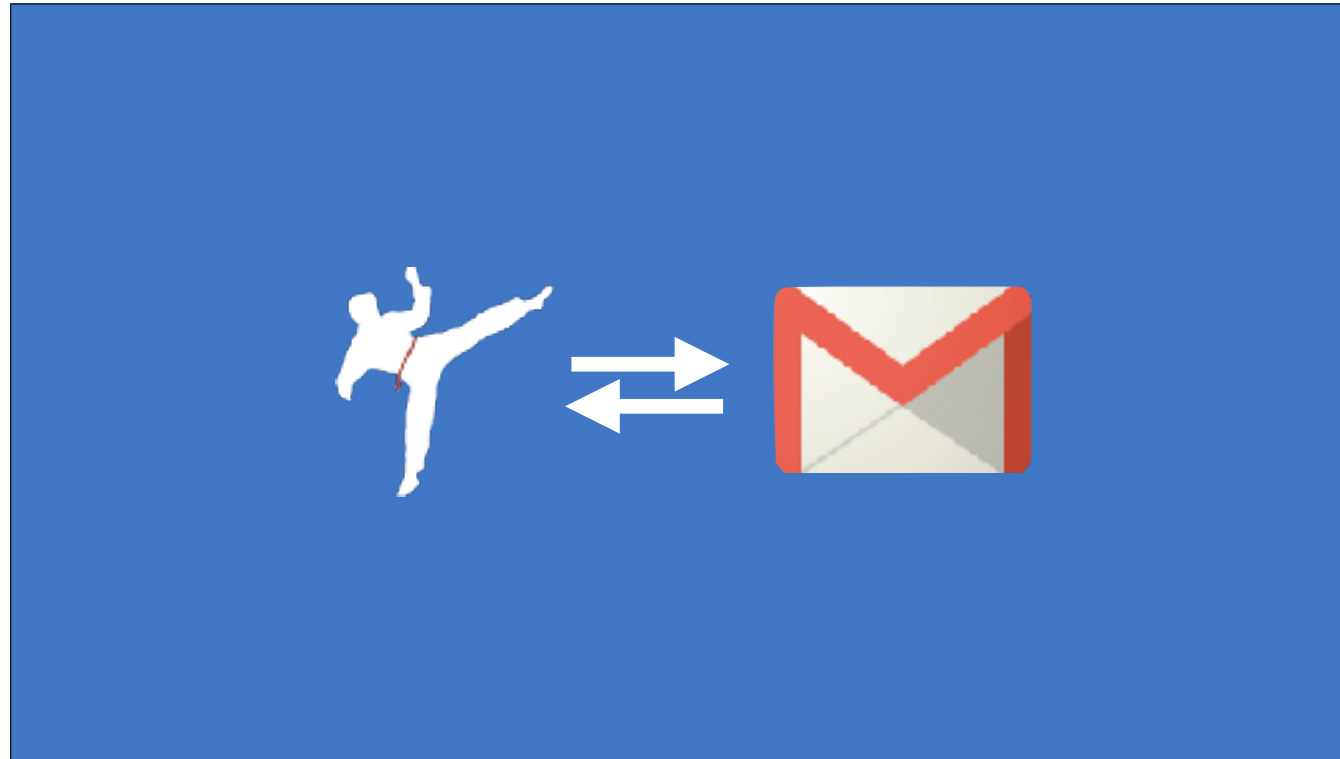
So after you signup through the browser, the server creates a User in the database for you, and sends you an email with some instructions. We can do it the way it's illustrated above and execute everything synchronously. But!



What if the mail fail to be sent for whatever reason? Should we fore the user to signup again? How do we cleanup our state, we have the user already in the database? Should we retry? If yes how many times? How long will the user wait?



To solve those problems we utilise a background worker. Or in simple words a detached process. In the case of the previous example, instead of sending out the mail immediately, we put it in a queue of things that need to be done eventually. In this case we use Sidekiq as our background worker solution.



Sidekiq will at some point read the job and send out the email. If it fails it will just retry until it succeeds. This is a huge plus!



???

But do we really need Sidekiq to do this?

```
Private | terra -> bunny_mail -> rdc -> insight -> zsh 213.202.169.34 192.336.113.241 %16/07/2016 100% 2018/02/26 21:52
1 require_relative 'chat'
2
3 class App < Node
4   Chat.start(:heartbeat)
5
6   plugin :streaming
7   plugin :render, engine: 'slim'
8
9   route do |r|
10     r.root do
11       r.get 'stream' do
12         response["Content-Type"] = "text/event-stream; charset=UTF-8"
13         user_queue = Chat.subscribe
14         callback = proc { Chat.unsubscribe(user_queue) }
15         stream :loop, true, :callback, callback do |out|
16           Chat.stream :out, user_queue
17         end
18       end
19     end
20
21     r.on "messages" do
22       r.post do
23         name = r.params['name']
24         message = r.params['message']
25         Chat.message name, message
26       end
27     end
28   end
29 end
30
31 end
32
33 end
34
35 end
36
37 end
38
39 end
40
41 end
42
43 end
44
45 end
46
47 end
48
49 end
50
51 end
52
53 end
54
55 end
56
57 end
58
59 end
60
61 end
62
63 end
64
65 end
66
67 end
68
69 end
70
71 end
72
73 end
74
75 end
76
77 end
78
79 end
80
81 end
82
83 end
84
85 end
86
87 end
88
89 end
90
91 end
92
93 end
94
95 end
96
97 end
98
99 end
100
101 end
102
103 end
104
105 end
106
107 end
108
109 end
110
111 end
112
113 end
114
115 end
116
117 end
118
119 end
120
121 end
122
123 end
124
125 end
126
127 end
128
129 end
130
131 end
132
133 end
134
135 end
136
137 end
138
139 end
140
141 end
142
143 end
144
145 end
146
147 end
148
149 end
150
151 end
152
153 end
154
155 end
156
157 end
158
159 end
160
161 end
162
163 end
164
165 end
166
167 end
168
169 end
170
171 end
172
173 end
174
175 end
176
177 end
178
179 end
180
181 end
182
183 end
184
185 end
186
187 end
188
189 end
190
191 end
192
193 end
194
195 end
196
197 end
198
199 end
200
201 end
202
203 end
204
205 end
206
207 end
208
209 end
210
211 end
212
213 end
214
215 end
216
217 end
218
219 end
220
221 end
222
223 end
224
225 end
226
227 end
228
229 end
230
231 end
232
233 end
234
235 end
236
237 end
238
239 end
240
241 end
242
243 end
244
245 end
246
247 end
248
249 end
250
251 end
252
253 end
254
255 end
256
257 end
258
259 end
260
261 end
262
263 end
264
265 end
266
267 end
268
269 end
270
271 end
272
273 end
274
275 end
276
277 end
278
279 end
280
281 end
282
283 end
284
285 end
286
287 end
288
289 end
290
291 end
292
293 end
294
295 end
296
297 end
298
299 end
300
301 end
302
303 end
304
305 end
306
307 end
308
309 end
310
311 end
312
313 end
314
315 end
316
317 end
318
319 end
320
321 end
322
323 end
324
325 end
326
327 end
328
329 end
330
331 end
332
333 end
334
335 end
336
337 end
338
339 end
340
341 end
342
343 end
344
345 end
346
347 end
348
349 end
350
351 end
352
353 end
354
355 end
356
357 end
358
359 end
360
361 end
362
363 end
364
365 end
366
367 end
368
369 end
370
371 end
372
373 end
374
375 end
376
377 end
378
379 end
380
381 end
382
383 end
384
385 end
386
387 end
388
389 end
390
391 end
392
393 end
394
395 end
396
397 end
398
399 end
400
401 end
402
403 end
404
405 end
406
407 end
408
409 end
410
411 end
412
413 end
414
415 end
416
417 end
418
419 end
420
421 end
422
423 end
424
425 end
426
427 end
428
429 end
430
431 end
432
433 end
434
435 end
436
437 end
438
439 end
440
441 end
442
443 end
444
445 end
446
447 end
448
449 end
450
451 end
452
453 end
454
455 end
456
457 end
458
459 end
460
461 end
462
463 end
464
465 end
466
467 end
468
469 end
470
471 end
472
473 end
474
475 end
476
477 end
478
479 end
480
481 end
482
483 end
484
485 end
486
487 end
488
489 end
490
491 end
492
493 end
494
495 end
496
497 end
498
499 end
500
501 end
502
503 end
504
505 end
506
507 end
508
509 end
510
511 end
512
513 end
514
515 end
516
517 end
518
519 end
520
521 end
522
523 end
524
525 end
526
527 end
528
529 end
530
531 end
532
533 end
534
535 end
536
537 end
538
539 end
540
541 end
542
543 end
544
545 end
546
547 end
548
549 end
550
551 end
552
553 end
554
555 end
556
557 end
558
559 end
560
561 end
562
563 end
564
565 end
566
567 end
568
569 end
570
571 end
572
573 end
574
575 end
576
577 end
578
579 end
580
581 end
582
583 end
584
585 end
586
587 end
588
589 end
590
591 end
592
593 end
594
595 end
596
597 end
598
599 end
600
601 end
602
603 end
604
605 end
606
607 end
608
609 end
610
611 end
612
613 end
614
615 end
616
617 end
618
619 end
620
621 end
622
623 end
624
625 end
626
627 end
628
629 end
630
631 end
632
633 end
634
635 end
636
637 end
638
639 end
640
641 end
642
643 end
644
645 end
646
647 end
648
649 end
650
651 end
652
653 end
654
655 end
656
657 end
658
659 end
660
661 end
662
663 end
664
665 end
666
667 end
668
669 end
670
671 end
672
673 end
674
675 end
676
677 end
678
679 end
680
681 end
682
683 end
684
685 end
686
687 end
688
689 end
690
691 end
692
693 end
694
695 end
696
697 end
698
699 end
700
701 end
702
703 end
704
705 end
706
707 end
708
709 end
710
711 end
712
713 end
714
715 end
716
717 end
718
719 end
720
721 end
722
723 end
724
725 end
726
727 end
728
729 end
730
731 end
732
733 end
734
735 end
736
737 end
738
739 end
740
741 end
742
743 end
744
745 end
746
747 end
748
749 end
750
751 end
752
753 end
754
755 end
756
757 end
758
759 end
760
761 end
762
763 end
764
765 end
766
767 end
768
769 end
770
771 end
772
773 end
774
775 end
776
777 end
778
779 end
780
781 end
782
783 end
784
785 end
786
787 end
788
789 end
790
791 end
792
793 end
794
795 end
796
797 end
798
799 end
800
801 end
802
803 end
804
805 end
806
807 end
808
809 end
810
811 end
812
813 end
814
815 end
816
817 end
818
819 end
820
821 end
822
823 end
824
825 end
826
827 end
828
829 end
830
831 end
832
833 end
834
835 end
836
837 end
838
839 end
840
841 end
842
843 end
844
845 end
846
847 end
848
849 end
850
851 end
852
853 end
854
855 end
856
857 end
858
859 end
860
861 end
862
863 end
864
865 end
866
867 end
868
869 end
870
871 end
872
873 end
874
875 end
876
877 end
878
879 end
880
881 end
882
883 end
884
885 end
886
887 end
888
889 end
890
891 end
892
893 end
894
895 end
896
897 end
898
899 end
900
901 end
902
903 end
904
905 end
906
907 end
908
909 end
910
911 end
912
913 end
914
915 end
916
917 end
918
919 end
920
921 end
922
923 end
924
925 end
926
927 end
928
929 end
930
931 end
932
933 end
934
935 end
936
937 end
938
939 end
940
941 end
942
943 end
944
945 end
946
947 end
948
949 end
950
951 end
952
953 end
954
955 end
956
957 end
958
959 end
960
961 end
962
963 end
964
965 end
966
967 end
968
969 end
970
971 end
972
973 end
974
975 end
976
977 end
978
979 end
980
981 end
982
983 end
984
985 end
986
987 end
988
989 end
990
991 end
992
993 end
994
995 end
996
997 end
998
999 end
1000
1001 end
1002
1003 end
1004
1005 end
1006
1007 end
1008
1009 end
1010
1011 end
1012
1013 end
1014
1015 end
1016
1017 end
1018
1019 end
1020
1021 end
1022
1023 end
1024
1025 end
1026
1027 end
1028
1029 end
1030
1031 end
1032
1033 end
1034
1035 end
1036
1037 end
1038
1039 end
1040
1041 end
1042
1043 end
1044
1045 end
1046
1047 end
1048
1049 end
1050
1051 end
1052
1053 end
1054
1055 end
1056
1057 end
1058
1059 end
1060
1061 end
1062
1063 end
1064
1065 end
1066
1067 end
1068
1069 end
1070
1071 end
1072
1073 end
1074
1075 end
1076
1077 end
1078
1079 end
1080
1081 end
1082
1083 end
1084
1085 end
1086
1087 end
1088
1089 end
1090
1091 end
1092
1093 end
1094
1095 end
1096
1097 end
1098
1099 end
1100
1101 end
1102
1103 end
1104
1105 end
1106
1107 end
1108
1109 end
1110
1111 end
1112
1113 end
1114
1115 end
1116
1117 end
1118
1119 end
1120
1121 end
1122
1123 end
1124
1125 end
1126
1127 end
1128
1129 end
1130
1131 end
1132
1133 end
1134
1135 end
1136
1137 end
1138
1139 end
1140
1141 end
1142
1143 end
1144
1145 end
1146
1147 end
1148
1149 end
1150
1151 end
1152
1153 end
1154
1155 end
1156
1157 end
1158
1159 end
1160
1161 end
1162
1163 end
1164
1165 end
1166
1167 end
1168
1169 end
1170
1171 end
1172
1173 end
1174
1175 end
1176
1177 end
1178
1179 end
1180
1181 end
1182
1183 end
1184
1185 end
1186
1187 end
1188
1189 end
1190
1191 end
1192
1193 end
1194
1195 end
1196
1197 end
1198
1199 end
1200
1201 end
1202
1203 end
1204
1205 end
1206
1207 end
1208
1209 end
1210
1211 end
1212
1213 end
1214
1215 end
1216
1217 end
1218
1219 end
1220
1221 end
1222
1223 end
1224
1225 end
1226
1227 end
1228
1229 end
1230
1231 end
1232
1233 end
1234
1235 end
1236
1237 end
1238
1239 end
1240
1241 end
1242
1243 end
1244
1245 end
1246
1247 end
1248
1249 end
1250
1251 end
1252
1253 end
1254
1255 end
1256
1257 end
1258
1259 end
1260
1261 end
1262
1263 end
1264
1265 end
1266
1267 end
1268
1269 end
1270
1271 end
1272
1273 end
1274
1275 end
1276
1277 end
1278
1279 end
1280
1281 end
1282
1283 end
1284
1285 end
1286
1287 end
1288
1289 end
1290
1291 end
1292
1293 end
1294
1295 end
1296
1297 end
1298
1299 end
1300
1301 end
1302
1303 end
1304
1305 end
1306
1307 end
1308
1309 end
1310
1311 end
1312
1313 end
1314
1315 end
1316
1317 end
1318
1319 end
1320
1321 end
1322
1323 end
1324
1325 end
1326
1327 end
1328
1329 end
1330
1331 end
1332
1333 end
1334
1335 end
1336
1337 end
1338
1339 end
1340
1341 end
1342
1343 end
1344
1345 end
1346
1347 end
1348
1349 end
1350
1351 end
1352
1353 end
1354
1355 end
1356
1357 end
1358
1359 end
1360
1361 end
1362
1363 end
1364
1365 end
1366
1367 end
1368
1369 end
1370
1371 end
1372
1373 end
1374
1375 end
1376
1377 end
1378
1379 end
1380
1381 end
1382
1383 end
1384
1385 end
1386
1387 end
1388
1389 end
1390
1391 end
1392
1393 end
1394
1395 end
1396
1397 end
1398
1399 end
1400
1401 end
1402
1403 end
1404
1405 end
1406
1407 end
1408
1409 end
1410
1411 end
1412
1413 end
1414
1415 end
1416
1417 end
1418
1419 end
1420
1421 end
1422
1423 end
1424
1425 end
1426
1427 end
1428
1429 end
1430
1431 end
1432
1433 end
1434
1435 end
1436
1437 end
1438
1439 end
1440
1441 end
1442
1443 end
1444
1445 end
1446
1447 end
1448
1449 end
1450
1451 end
1452
1453 end
1454
1455 end
1456
1457 end
1458
1459 end
1460
1461 end
1462
1463 end
1464
1465 end
1466
1467 end
1468
1469 end
1470
1471 end
1472
1473 end
1474
1475 end
1476
1477 end
1478
1479 end
1480
1481 end
1482
1483 end
1484
1485 end
1486
1487 end
1488
1489 end
1490
1491 end
1492
1493 end
1494
1495 end
1496
1497 end
1498
1499 end
1500
1501 end
1502
1503 end
1504
1505 end
1506
1507 end
1508
1509 end
1510
1511 end
1512
1513 end
1514
1515 end
1516
1517 end
1518
1519 end
1520
1521 end
1522
1523 end
1524
1525 end
1526
1527 end
1528
1529 end
1530
1531 end
1532
1533 end
1534
1535 end
1536
1537 end
1538
1539 end
1540
1541 end
1542
1543 end
1544
1545 end
1546
1547 end
1548
1549 end
1550
1551 end
1552
1553 end
1554
1555 end
1556
1557 end
1558
1559 end
1560
1561 end
1562
1563 end
1564
1565 end
1566
1567 end
1568
1569 end
1570
1571 end
1572
1573 end
1574
1575 end
1576
1577 end
1578
1579 end
1580
1581 end
1582
1583 end
1584
1585 end
1586
1587 end
1588
1589 end
1590
1591 end
1592
1593 end
1594
1595 end
1596
1597 end
1598
1599 end
1600
1601 end
1602
1603 end
1604
1605 end
1606
1607 end
1608
1609 end
1610
1611 end
1612
1613 end
1614
1615 end
1616
1617 end
1618
1619 end
1620
1621 end
1622
1623 end
1624
1625 end
1626
1627 end
1628
1629 end
1630
1631 end
1632
1633 end
1634
1635 end
1636
1637 end
1638
1639 end
1640
1641 end
1642
1643 end
1644
1645 end
1646
1647 end
1648
1649 end
1650
1651 end
1652
1653 end
1654
1655 end
1656
1657 end
1658
1659 end
1660
1661 end
1662
1663 end
1664
1665 end
1666
1667 end
1668
1669 end
1670
1671 end
1672
1673 end
1674
1675 end
1676
1677 end
1678
1679 end
1680
1681 end
1682
1683 end
1684
1685 end
1686
1687 end
1688
1689 end
1690
1691 end
1692
1693 end
1694
1695 end
1696
1697 end
1698
1699 end
1700
1701 end
1702
1703 end
1704
1705 end
1706
1707 end
1708
1709 end
1710
1711 end
1712
1713 end
1714
1715 end
1716
1717 end
1718
1719 end
1720
1721 end
1722
1723 end
1724
1725 end
1726
1727 end
1728
1729 end
1730
1731 end
1732
1733 end
1734
1735 end
1736
1737 end
1738
1739 end
1740
1741 end
1742
1743 end
1744
1745 end
1746
1747 end
1748
1749 end
1750
1751 end
1752
1753 end
1754
1755 end
1756
1757 end
1758
1759 end
1760
1761 end
1762
1763 end
1764
1765 end
1766
1767 end
1768
1769 end
1770
1771 end
1772
1773 end
1774
1775 end
1776
1777 end
1778
1779 end
1780
1781 end
1782
1783 end
1784
1785 end
1786
1787 end
1788
1789 end
1790
1791 end
1792
1793 end
1794
1795 end
1796
1797 end
1798
1799 end
1800
1801 end
1802
1803 end
1804
1805 end
1806
1807 end
1808
1809 end
1810
1811 end
1812
1813 end
1814
1815 end
1816
1817 end
1818
1819 end
1820
1821 end
1822
1823 end
1824
1825 end
1826
1827 end
1828
1829 end
1830
1831 end
1832
1833 end
1834
1835 end
1836
1837 end
1838
1839 end
1840
1841 end
1842
1843 end
1844
1845 end
1846
1847 end
1848
1849 end
1850
1851 end
1852
1853 end
1854
1855 end
1856
1857 end
1858
1859 end
1860
1861 end
1862
1863 end
1864
1865 end
1866
1867 end
1868
1869 end
1870
1871 end
1872
1873 end
1874
1875 end
1876
1877 end
1878
1879 end
1880
1881 end
1882
1883 end
1884
1885 end
1886
1887 end
1888
1889 end
1890
1891 end
1892
1893 end
1894
1895 end
1896
1897 end
1898
1899 end
1900
1901 end
1902
1903 end
1904
1905 end
1906
1907 end
1908
1909 end
1910
1911 end
1912
1913 end
1914
1915 end
1916
1917 end
1918
1919 end
1920
1921 end
1922
1923 end
1924
1925 end
1926
1927 end
1928
1929 end
1930
1931 end
1932
1933 end
1934
1935 end
1936
1937 end
1938
1939 end
1940
1941 end
1942
1943 end
1944
1945 end
1946
1947 end
1948
1949 end
1950
1951 end
1952
1953 end
1954
1955 end
1956
1957 end
1958
1959 end
1960
1961 end
1962
1963 end
1964
1965 end
1966
1967 end
1968
1969 end
1970
1971 end
1972
1973 end
1974
1975 end
1976
1977 end
1978
1979 end
1980
1981 end
1982
1983 end
1984
1985 end
1986
1987 end
1988
1989 end
1990
1991 end
1992
1993 end
1994
1995 end
1996
1997 end
1998
1999 end
2000
2001 end
2002
2003 end
2004
2005 end
2006
2007 end
2008
2009 end
2010
2011 end
2012
2013 end
2014
2015 end
2016
2017 end
2018
2019 end
2020
2021 end
2022
2023 end
2024
2025 end
2026
2027 end
2028
2029 end
2030
2031 end
2032
2033 end
2034
2035 end
2036
2037 end
2038
2039 end
2040
2041 end
2042
2043 end
2044
2045 end
2046
2047 end
2048
2049 end
2050
2051 end
2052
2053 end
2054
2055 end
2056
2057 end
2058
2059 end
2060
2061 end
2062
2063 end
2064
2065 end
2066
2067 end
2068
2069 end
2070
2071 end
2072
2073 end
2074
2075 end
2076
2077 end
2078
2079 end
2080
2081 end
2082
2083 end
2084
2085 end
2086
2087 end
2088
2089 end
2090
2091 end
2092
2093 end
2094
2095 end
2096
2097 end
2098
2099 end
2100
2101 end
2102
2103 end
2104
2105 end
2106
2107 end
2108
2109 end
2110
2111 end
2112
2113 end
2114
2115 end
2116
2117 end
2118
2119 end
2120
2121 end
2122
2123 end
2124
2125 end
2126
2127 end
2128
2129 end
2130
2131 end
2132
2133 end
2134
2135 end
2136
2137 end
2138
2139 end
2140
2141 end
2142
2143 end
2144
2145 end
2146
2147 end
2148
2149 end
2150
2151 end
2152
2153 end
2154
2155 end
2156
2157 end
2158
2159 end
2160
2161 end
2162
2163 end
2164
2165 end
2166
2167 end
2168
2169 end
2170
2171 end
2172
2173 end
2174
2175 end
2176
2177 end
2178
2179 end
2180
2181 end
2182
2183 end
2184
2185 end
2186
2187 end
2188
2189 end
2190
2191 end
2192
2193 end
2194
2195 end
2196
2197 end
2198
2199 end
2200
2201 end
2202
2203 end
2204
2205 end
2206
2207 end
2208
2209 end
2210
2211 end
2212
2213 end
2214
2215 end
2216
2217 end
2218
2219 end
2220
2221 end
2222
2223 end
2224
2225 end
2226
2227 end
2228
2229 end
2230
2231 end
2232
2233 end
2234
2235 end
2236
2237 end
2238
2239 end
2240
2241 end
2242
2243 end
2244
2245 end
2246
2247 end
2248
2249 end
2250
2251 end
2252
2253 end
2254
2255 end
2256
2257 end
2258
2259 end
2260
2261 end
2262
2263 end
2264
2265 end
2266
2267 end
2268
2269 end
2270
2271 end
2272
2273 end
2274
2275 end
2276
2277 end
2278
2279 end
2280
2281 end
2282
2283 end
2284
2285 end
2286
2287 end
2288
2289 end
2290
2291 end
2292
2293 end
2294
2295 end
2296
2297 end
2298
2299 end
2300
2301 end
2302
2303 end
2304
2305 end
2306
2307 end
2308
2309 end
2310
2311 end
2312
2313 end
2314
2315 end
2316
2317 end
2318
2319 end
2320
2321 end
2322
2323 end
2324
2325 end
2326
2327 end
2328
2329 end
2330
2331 end
2332
2333 end
2334
2335 end
2336
2337 end
2338
2339 end
2340
2341 end
2342
2343 end
2344
2345 end
2346
2347 end
2348
2349 end
2350
2351 end
2352
2353 end
2354
2355 end
2356
2357 end
2358
2359 end
2360
2361 end
2362
2363 end
2364
2365 end
2366
2367 end
2368
2369 end
2370
2371 end
2372
2373 end
2374
2375 end
2376
2377 end
2378
2379 end
2380
2381 end
2382
2383 end
2384
2385 end
2386
2387 end
2388
2389 end
2390
2391 end
2392
2393 end
2394
2395 end
2396
2397 end
2398
2399 end
2400
2401 end
2402
2403 end
2404
2405 end
2406

```




Downsides?

Ok... So why do people use Sidekiq if this is so easy to implement in plain Ruby?

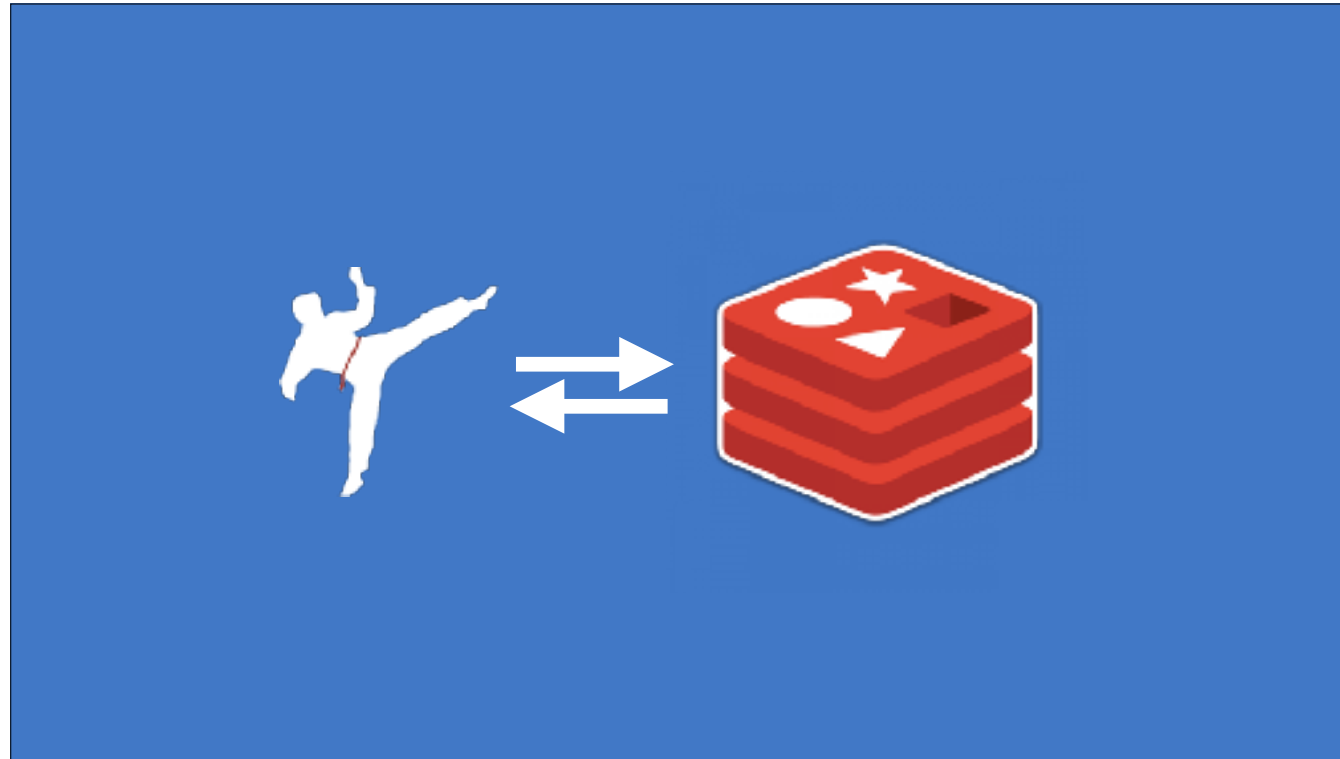
Debugging

Well, this is hard to debug. It's not exactly trivial to inspect the contents of the queue.

Load balancing

On the other hand this queue is Ruby prices specific. If I create 10 instances of this app each would have it's own queue and it's own workers. If one is shot down it's jobs are lost.

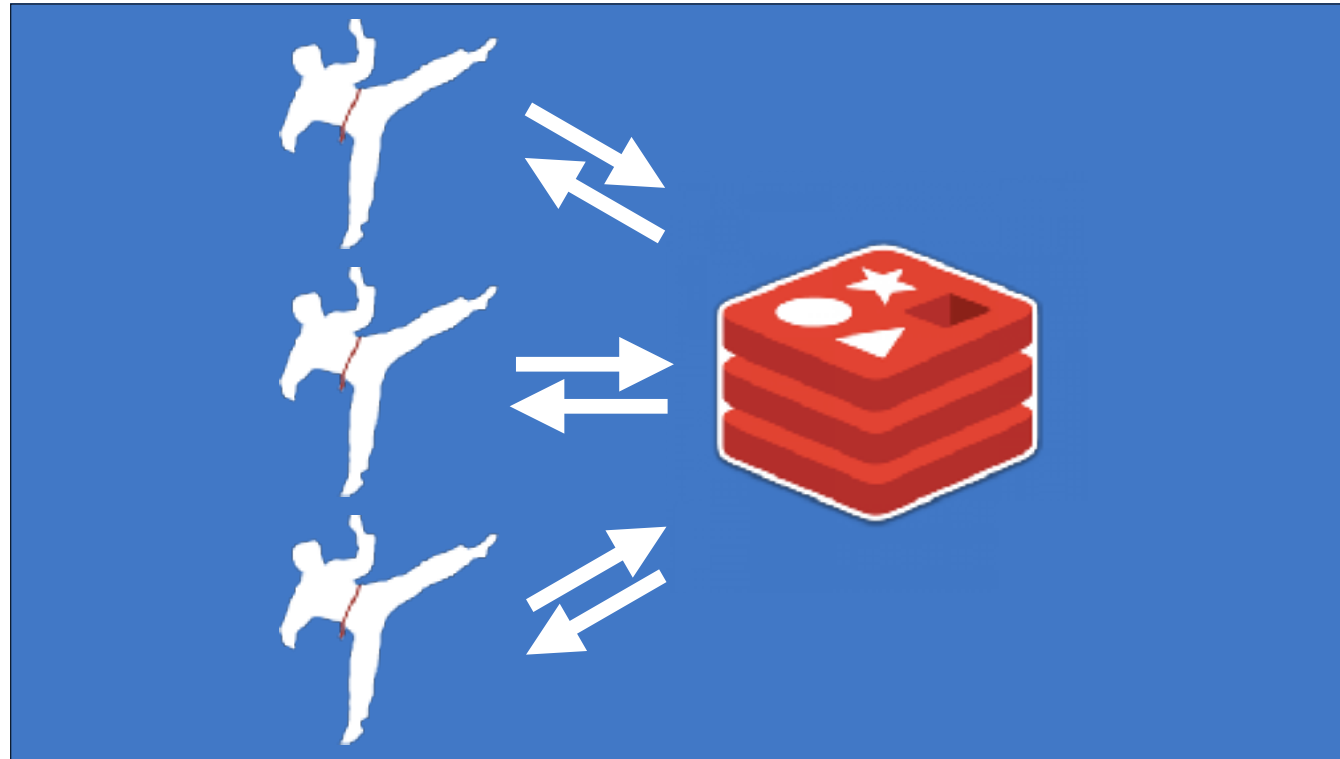
Note that this isn't impossible to implement, but it would quickly spiral into a project of it's own.



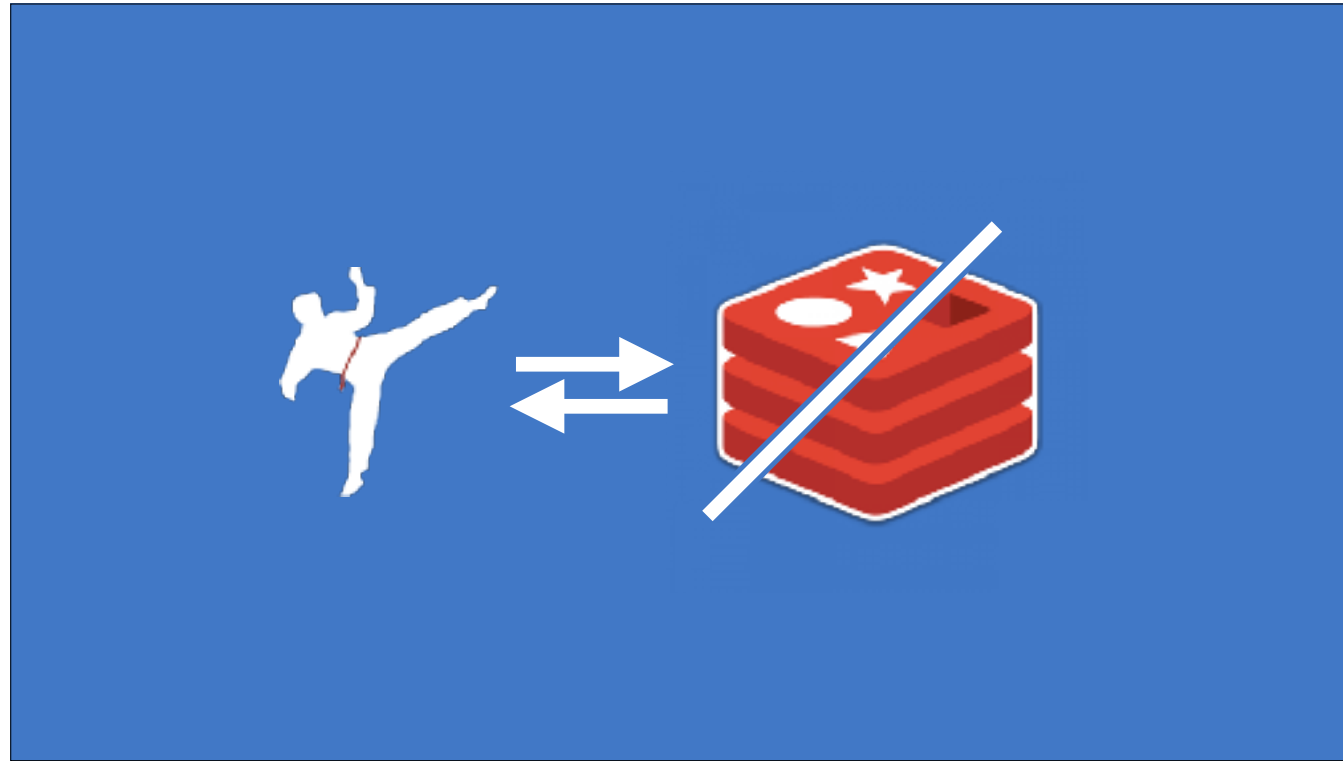
To solve those issues Sidekiq uses Redis as a backend (database).

```
FloatinPoint 1 > papa
127.0.0.1:6379> KEYS +
1) "stat:failed:push_notify"
2) "dead"
3) "stat:failed"
4) "retry"
127.0.0.1:6379> ZRANGE retry 0 -1
1) [{"retry_count":10,"failed_at":"1519577858","queue":"push_notify","max_retry_count":10,"id":"020056876190583900","finished_at":null,"failed_at":"1520668867","error_message":"no match on right hand side value: nil","error_backtrace":""} (papa_pal) lib/papa_pal/workers/push_apns.ecs:9: PapaPal.Workers.PushApns.perform/3\\n    (werk) lib/work/worker.ex:35: Work.Worker.handle_cast/2\\n    (stdlib) gen_server.erl:606: gen_server.try_dispatch/4\\n    (stdlib) gen_server.erl:686: gen_server.handle_msg/6\\n    (stdlib) proc_lib.erl:247: proc_lib:init_p.do_apply/2\\n\\n    "enqueued_at":"1519566066","class":"PapaPal.Workers.PushApns","args":["92018ef-2bc3-44e1-ad8c-fa1a2402762a"],"Remember to complete your visit":"https://papa.ltd/"} null]
127.0.0.1:6379>
```

The queue is stored as a sorted set of JSON objects. It's quite easy to check the contents of the queue.



And since there is one central database of all the jobs, we can have multiple instances consume the same job queue.



But what happens if Redis fails? If configured to persist all changes to disk and keep a journal, nothing will happen. The second that Redis comes back online, most likely everything will be working as normal. But this mechanisms come at a cost!

RDB disadvantages

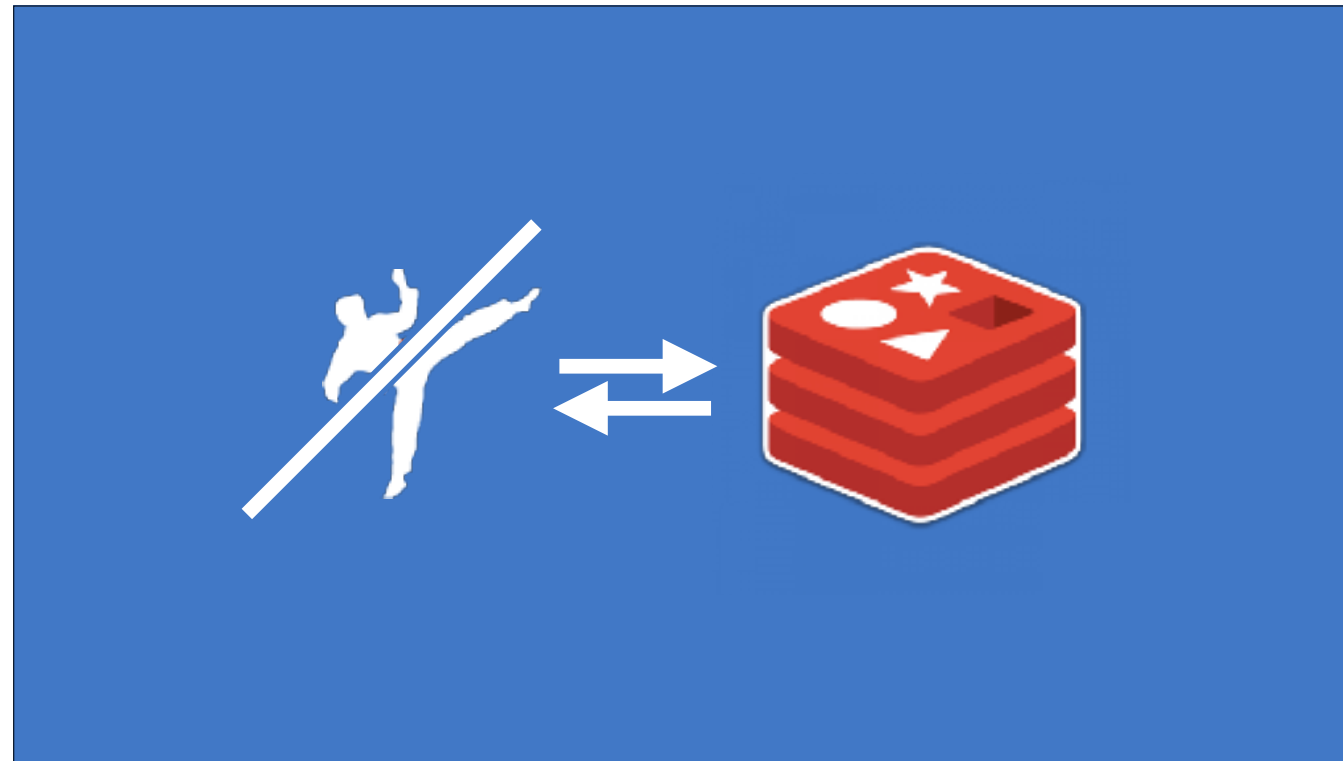
- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

If you opt to keep a transactional journal, be prepared to lose some data and to have slow connections.

AOF disadvantages

- AOF files are usually bigger than the equivalent RDB files for the same dataset.
- **AOF can be slower than RDB depending on the exact fsync policy.** In general with fsync set to *every second* performances are still very high, and with fsync disabled it should be exactly as fast as RDB even under high load. **Still RDB is able to provide more guarantees about the maximum latency even in the case of an huge write load.**
- In the past we experienced rare bugs in specific commands (for instance there was one involving blocking commands like BRPOPLPUSH) causing the AOF produced to not reproduce exactly the same dataset on reloading. This bugs are rare and we have tests in the test suite creating random complex datasets automatically and reloading them to check everything is ok, but this kind of bugs are almost impossible with RDB persistence. To make this point more clear: the Redis AOF works incrementally updating an existing state, like MySQL or MongoDB does, while the RDB snapshotting creates everything from scratch again and again, that is conceptually more robust. However - 1) It should be noted that every time the AOF is rewritten by Redis it is recreated from scratch starting from the actual data contained in the data set, making resistance to bugs stronger compared to an always appending AOF file (or one rewritten reading the old AOF instead of reading the data in memory). 2) We never had a single report from users about an AOF corruption that was detected in the real world.

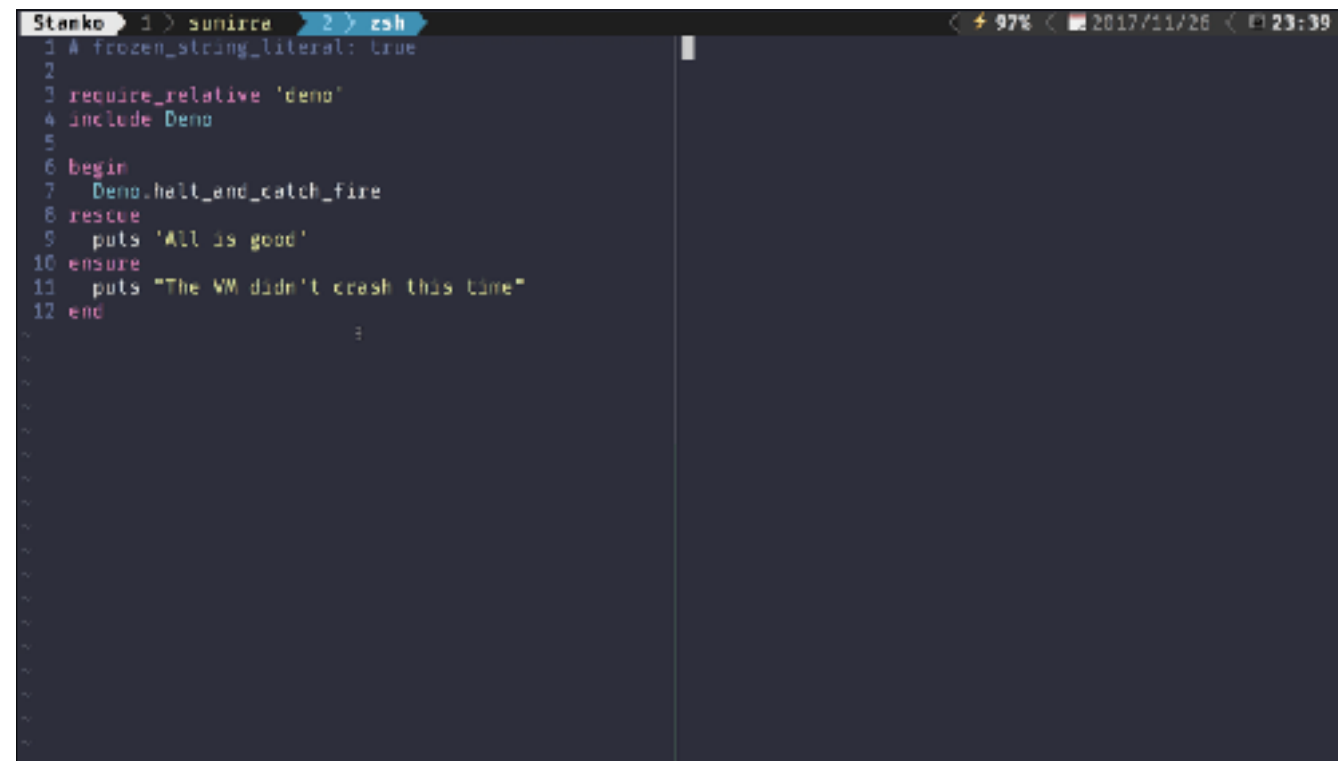
And if you opt for data persistence on disk be prepared to experience performance fluctuations.



Now we come to the real issues. What happens if your worker fails?

```
1 class TestWorker
2   include Sidekiq::Worker
3
4   def perform(*args)
5     # Do something
6   rescue
7     # Do something else
8   end
9 end
```

Well... We can always use Ruby to resolve exceptions! In fact, Sidekiq already does this for us! And it offers retry strategies, and exponential backoff and other bells and whistles!



A screenshot of a Stanko terminal window. The title bar shows 'Stanko' and two tabs: '1 > sunirca' and '2 > zsh'. The status bar on the right indicates 97% battery, the date 2017/11/26, and the time 23:39. The terminal content shows a Ruby script with the following lines:

```
1 # frozen_string_literal: true
2
3 require_relative 'deno'
4 include Deno
5
6 begin
7   Deno.halt_and_catch_fire
8 rescue
9   puts 'All is good'
10 ensure
11   puts "The VM didn't crash this time"
12 end
```

Below line 12, there are several lines of text that are mostly obscured by a vertical scrollbar on the right, but some characters like '~' and '=' are visible.

Well... No, not really. There are some exceptions that simply cant be caught.

Process Crashes

If the Sidekiq process segfaults or crashes the Ruby VM, any jobs that were being processed are lost. [Sidekiq Pro](#) offers a [reliable queueing](#) feature which does not lose those jobs.

Note that Sidekiq does offer protection against those kinds of errors, though it doesn't specify how it does so.

Sidekiq Simplifies Mail Background Processing for Ruby

HOME FEATURES PRICING ABOUT

OSS	PRO	ENTERPRISE
AUDIENCE	Multi-User	Multi-User/Enterprise
EXTENSIBLE	None	Small
USERS	Unlimited	Unlimited
PRICE	Free	\$1500 / year
FEATURES	None	Unlimited

GET STARTED >

LEARN MORE >

LEARN MORE >

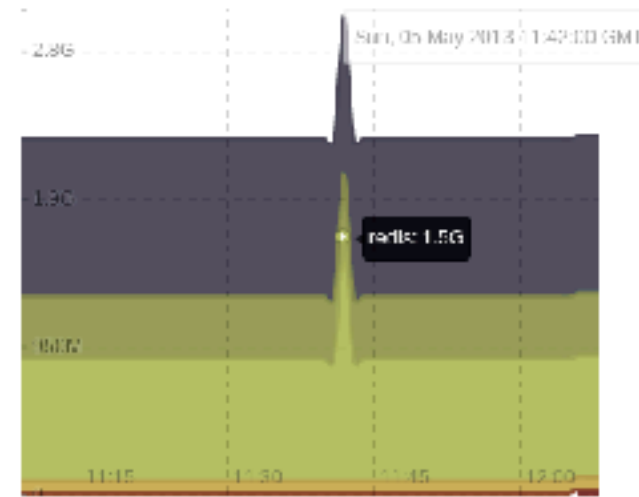
BUY >

BUY >

\$950 / year
Unlimited Use


And that feature comes with a \$1000 price tag. But even the \$2000 version won't save you from the following.

proc mem prod



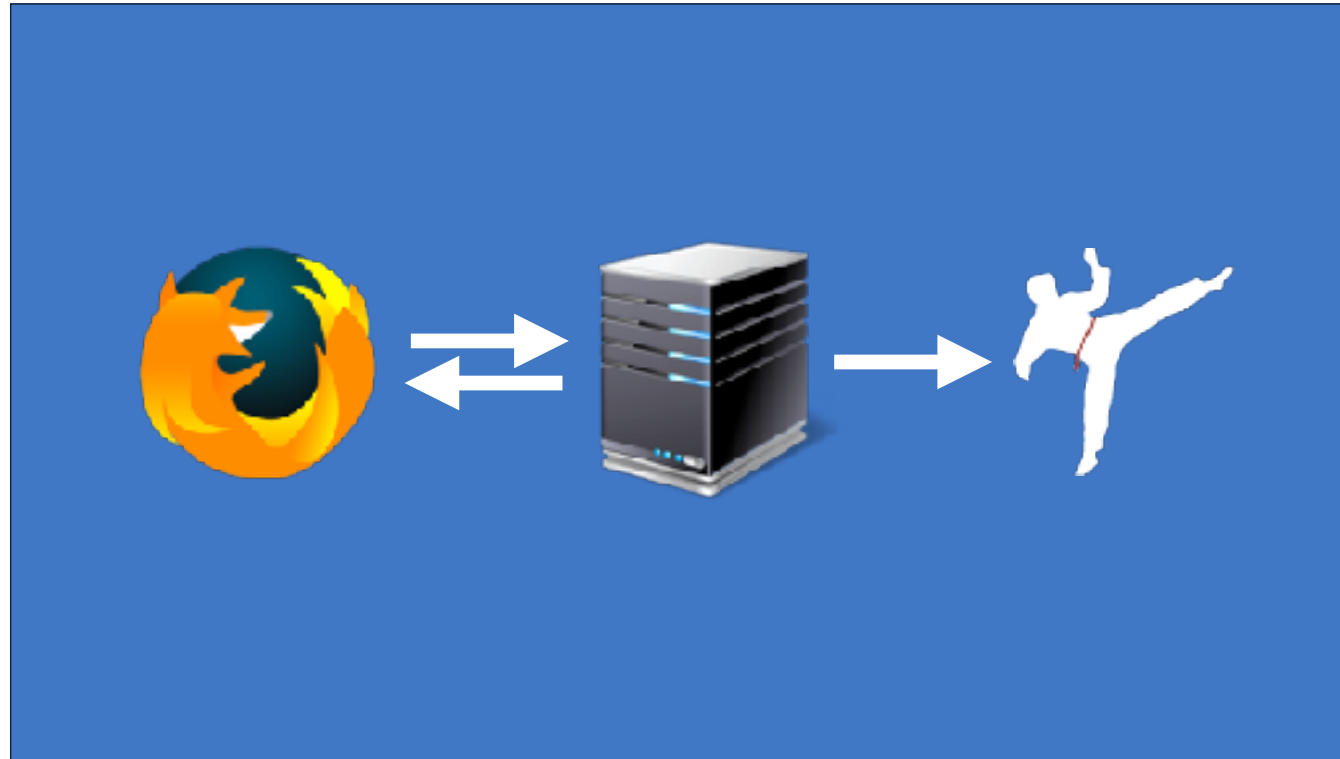
main process memory usage on production server

Memory consumption! Redis is an in-memory database, which means that it keeps all data in-memory. If you pass it 1.5GB of data, it will store that data in-memory. Even with persistence turned on, everything is still stored in-memory.

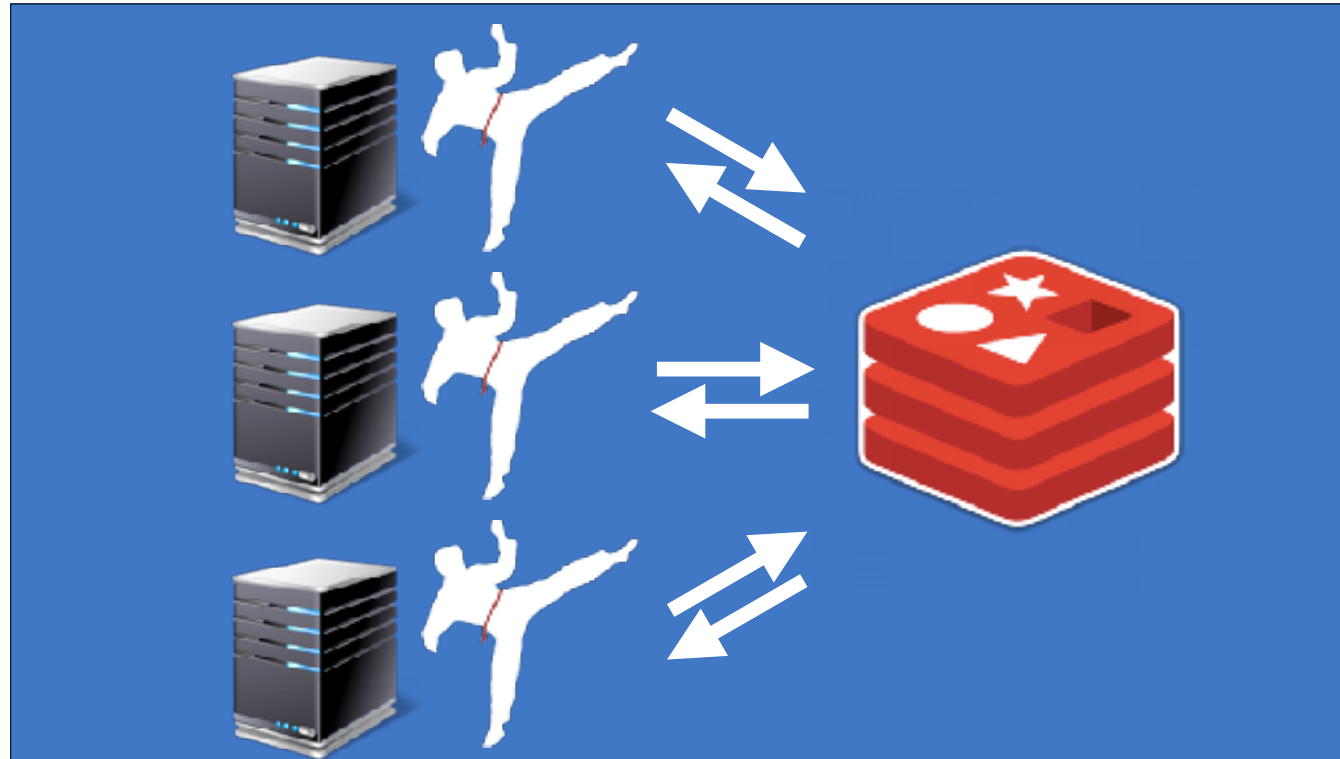


???

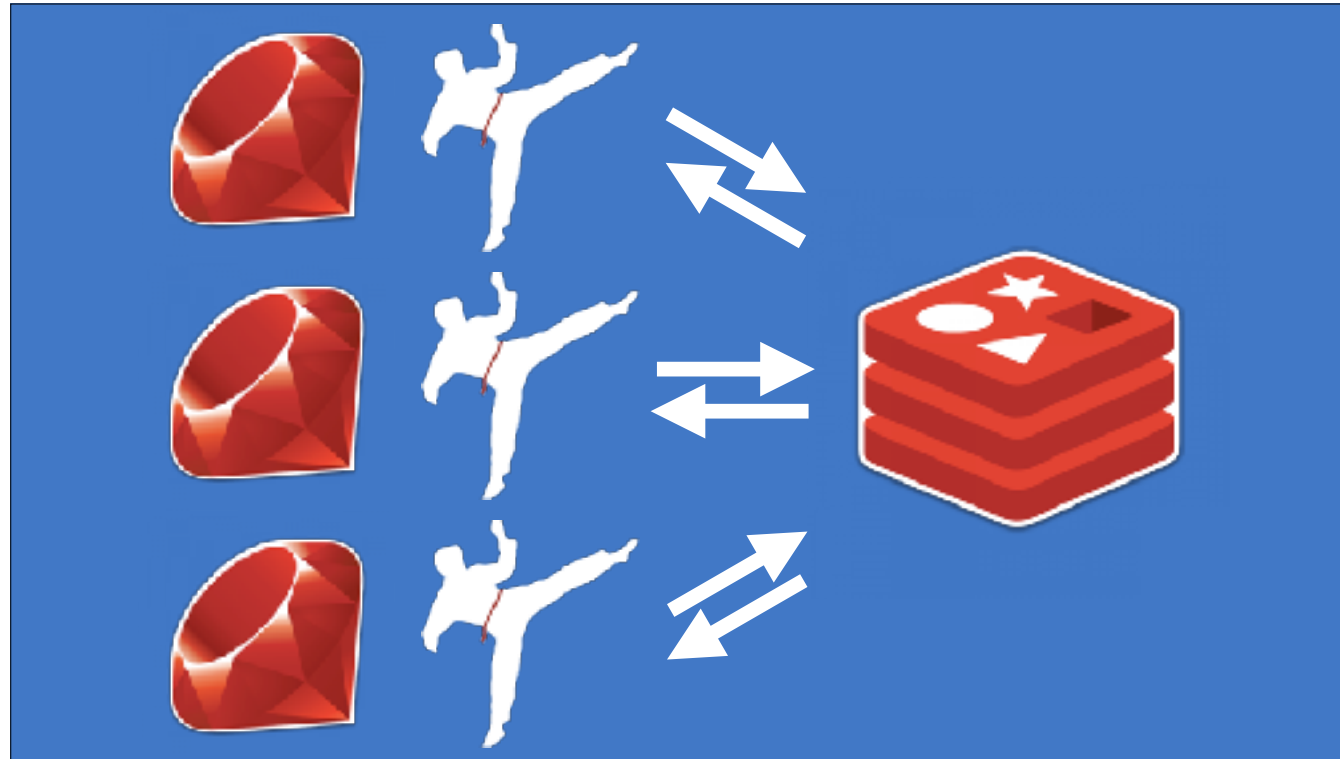
Ok, that can be attributed to bad design. Somebody is passing whole objects instead of only their IDs. In that case we have a potential scaling problem.



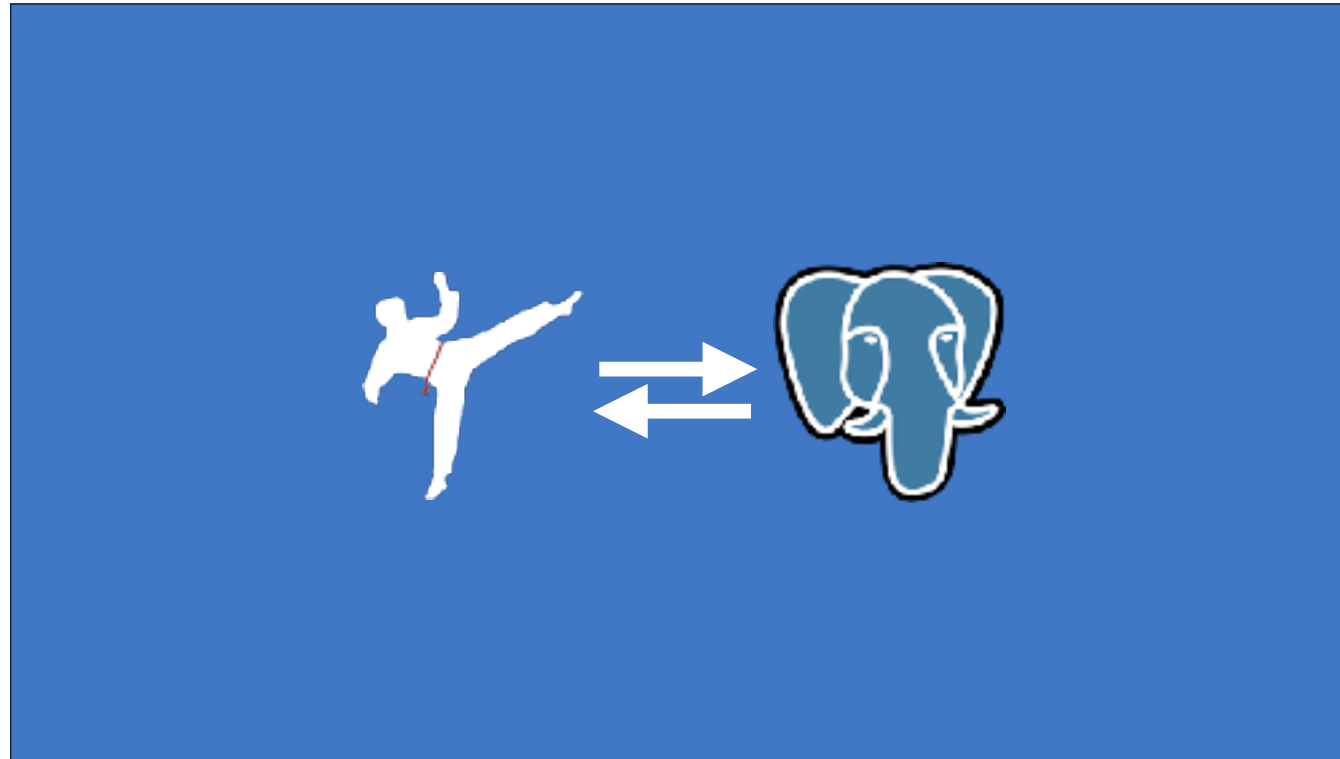
While our web app seemingly talks to a single Sidekiq instance.



Each redis worker keeps an instance of the whole application in memory. This is the most common case I have seen while working with Rails applications.



To solve this problem we can start Sidekiq workers as separate light-weight processes. This would solve the memory consumption issue.



But do we then give the workers direct access to the database? If yes then we need to manage models in two separate apps.



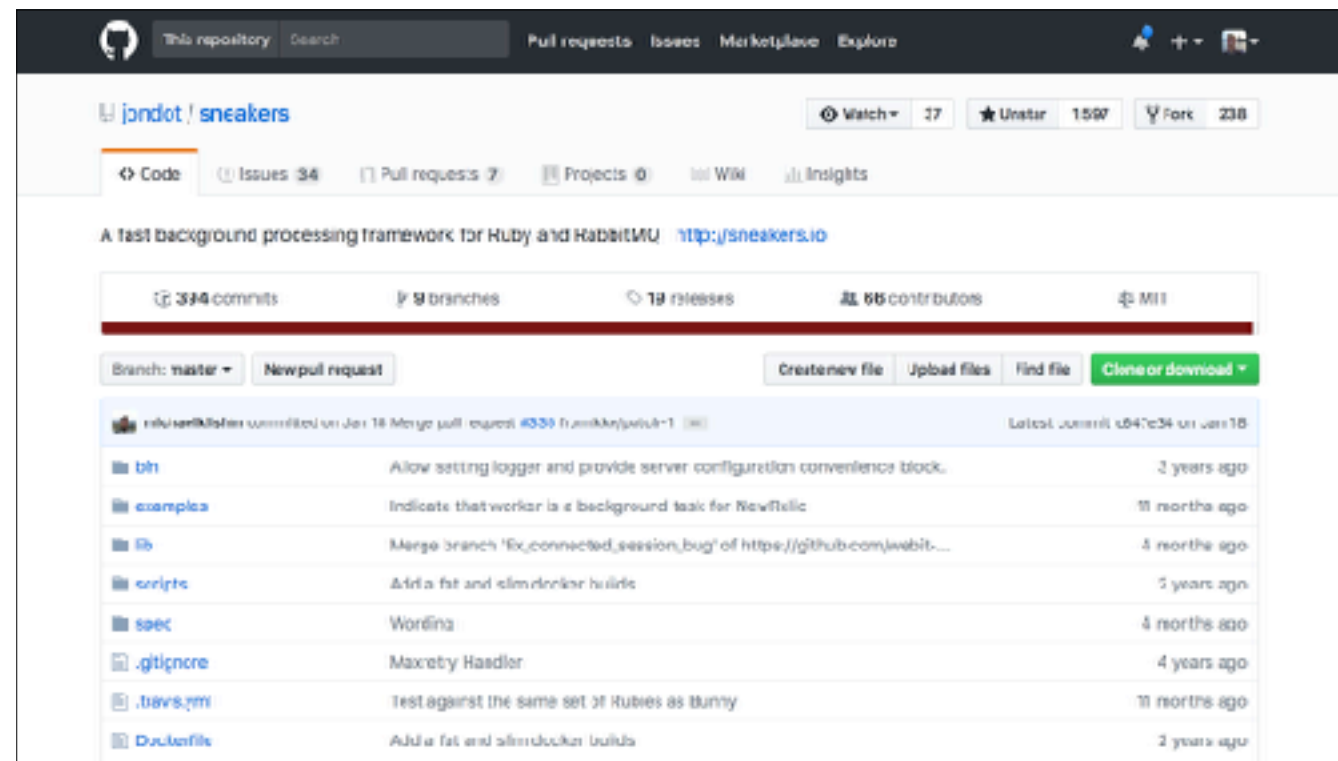
If no then we need to store and retrieve data through our own app, thus increasing load and potentially introducing temporary data do our DB.



This brings us to RabbitMQ.



RabbitMQ is a lot more than a simple job queue, but I'll start with that utility of it, since it's the one directly comparable to Sidekiq.



To use it as a background worker I would highly recommend the Sneakers project.

```

1 class TitleScraper
2   include Sneakers::Worker
3
4   from_queue 'downloads'
5
6   def work(msg)
7     title = extract_title(msg)
8     logger.info "FOUND <#{title}>"
9     ack!
10  end
11
12  private
13
14  def extract_title(html)
15    html.scan(/<title>(.*?)</title>/).flatten.first
16  end
17 end

```

```

1 class TestWorker
2   include Sidekiq::Worker
3
4   def perform(*args)
5     # Do something
6   end
7 end
8
9
10
11
12
13
14
15
16
17

```

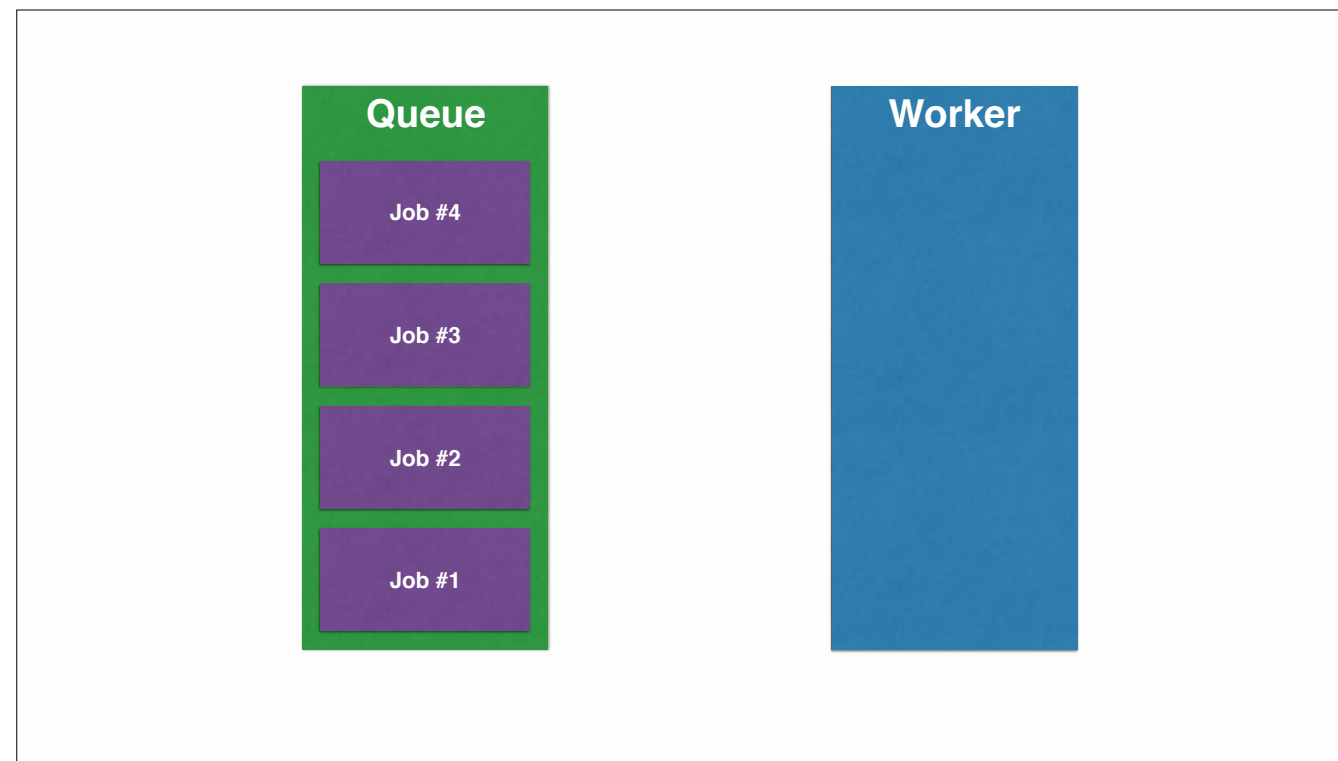
You would use it exactly the same as you would use Sidekiq. Even the syntax is similar. On the left is a Sneakers worker, on the right is a Sidekiq worker.

```
1 class TitleScraper
2   include Sneakers::Worker
3
4   from_queue 'downloads'
5
6   def work(msg)
7     title = extract_title(msg)
8     logger.info "FOUND <#{title}>"
9     ack!
10  end
11
12  private
13
14  def extract_title(html)
15    html.scan(/<title>(.*?)<\/title>/).flatten.first
16  end
17 end
```

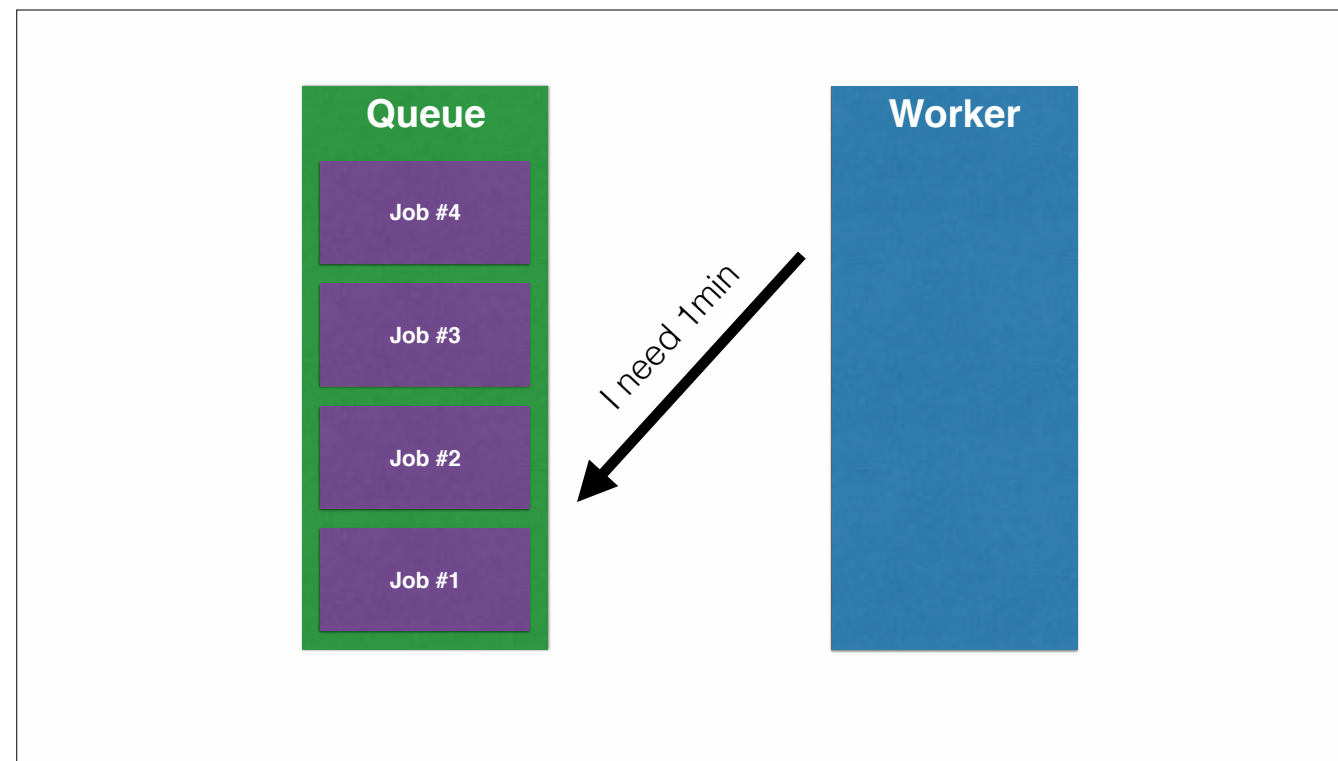
Here we can also notice a big feature of RabbitMQ and it's underlying protocol AMPQ

```
1 class TitleScraper
2   include Sneakers::Worker
3
4   from_queue 'downloads'
5
6   def work(msg)
7     title = extract_title(msg)
8     logger.info "FOUND <#{title}>"
9     ack!
10  end
11
12  private
13
14  def extract_title(html)
15    html.scan(/<title>(.*?)<\/title>/).flatten.first
16  end
17 end
```

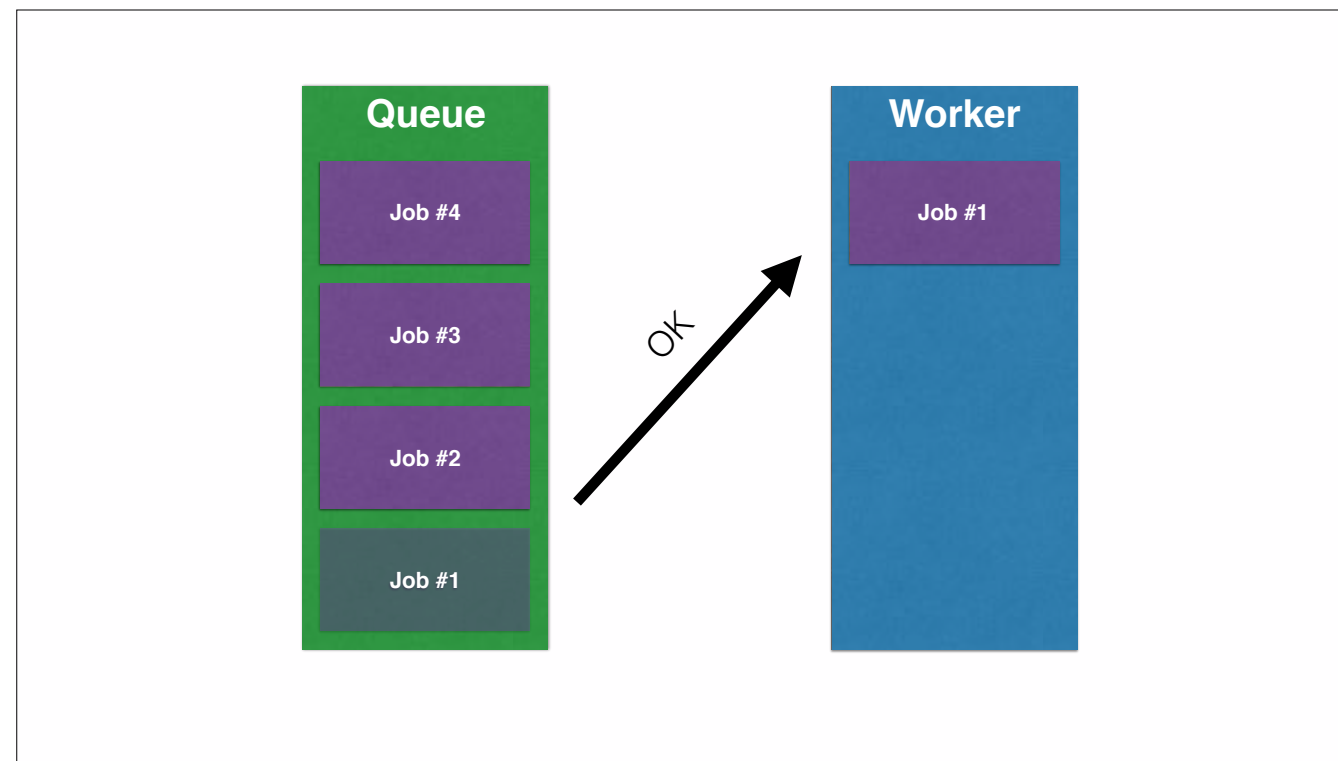
This little `ack!` method guarantees that the job will be processed no matter if RabbitMQ or the worker fail.



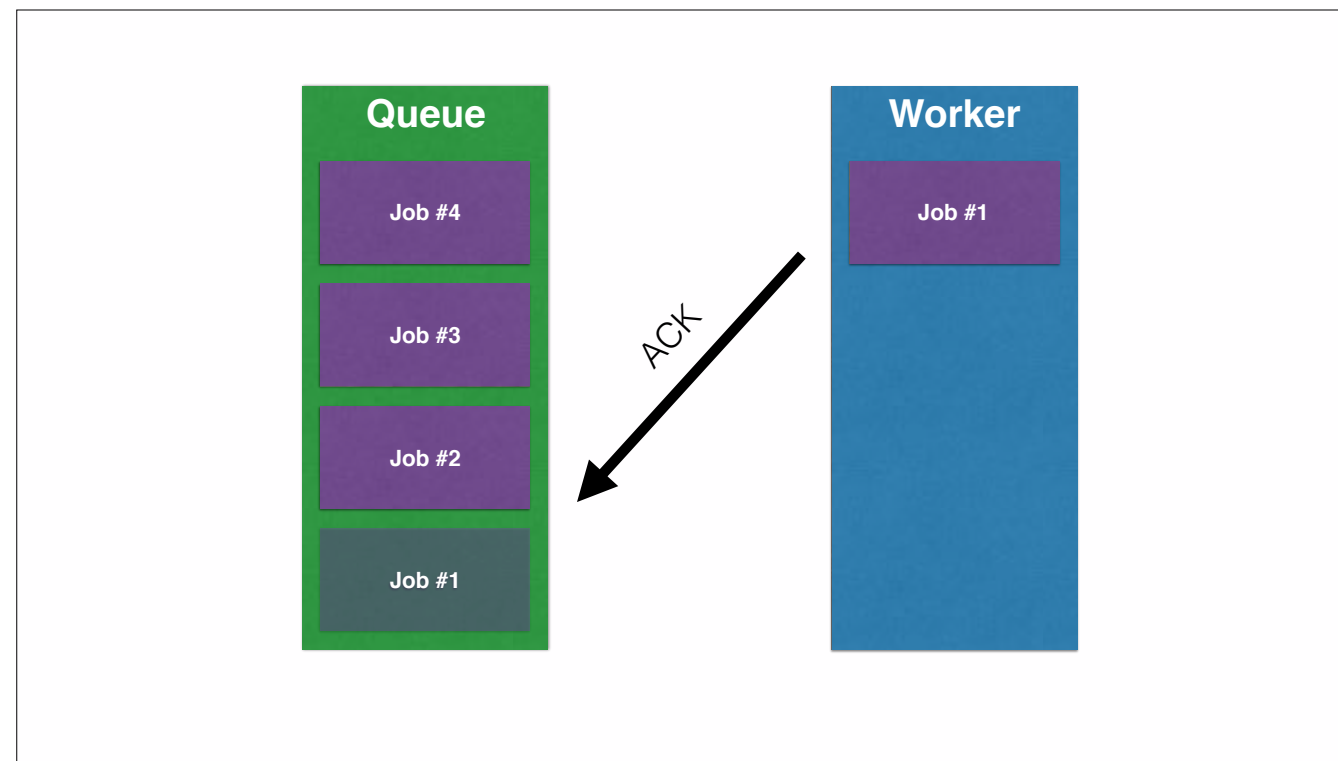
AMQP has two ways of fetching data from a queue. One is ACK mode and the other is NO-ACK mode. Here is an example of how ACK mode works.



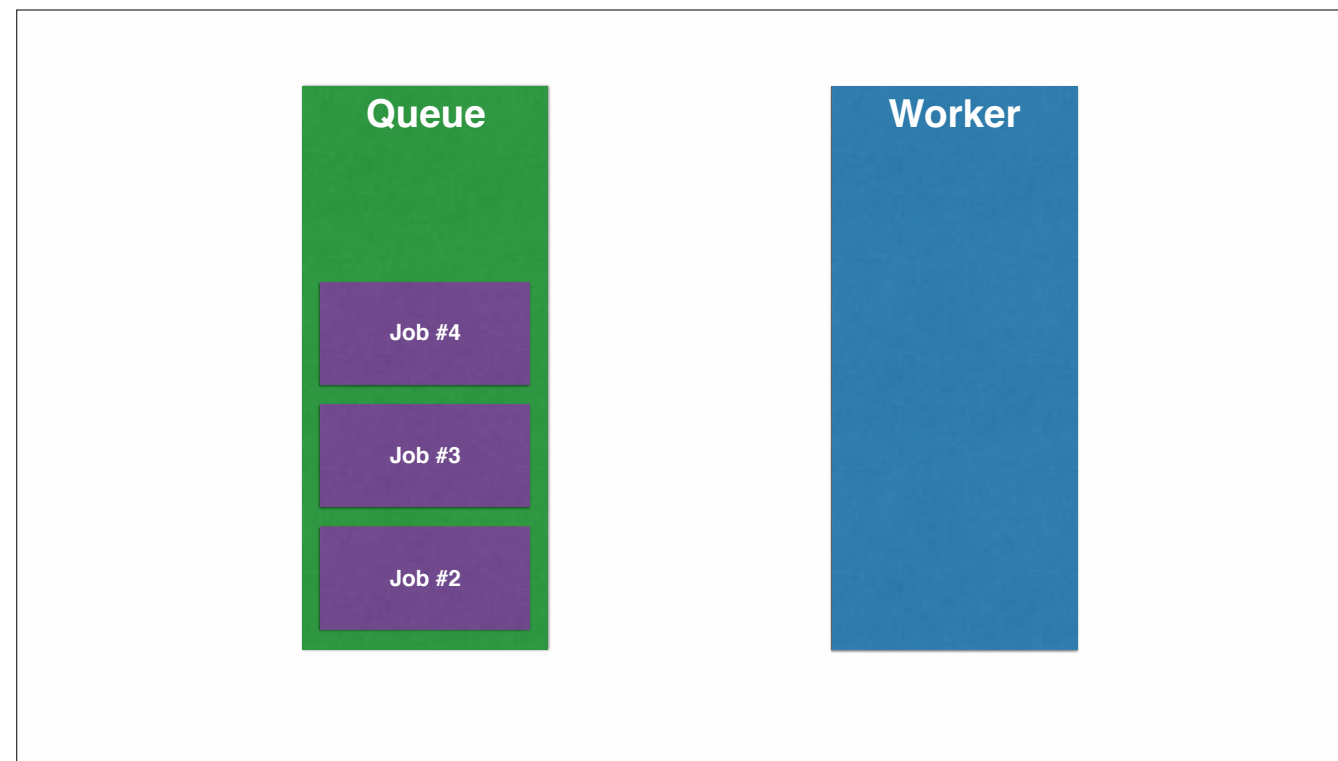
A worker, before popping a queue first specifies how long it will take to process the job. In this case 1min.



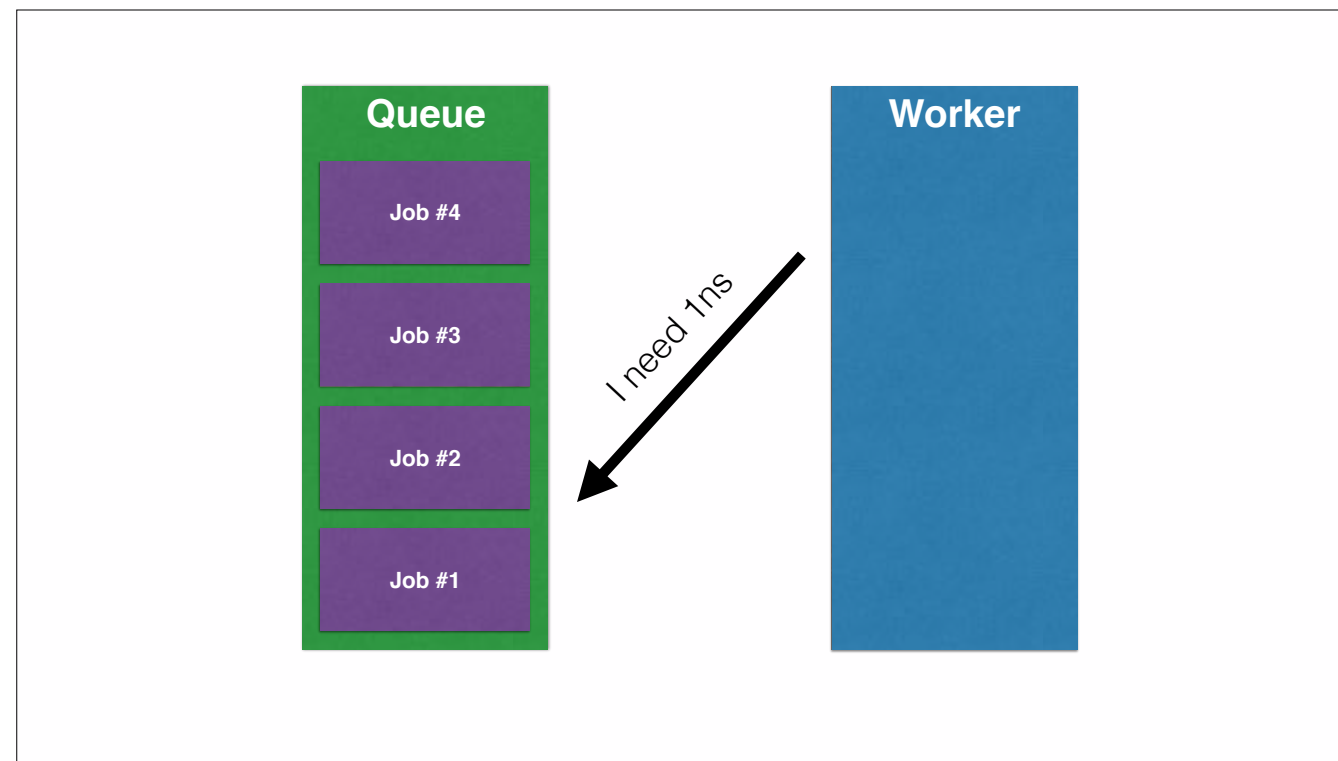
At that point the job is virtually removed from the queue, and the worker starts processing it. This job is still in the queue, but other workers can't see it!



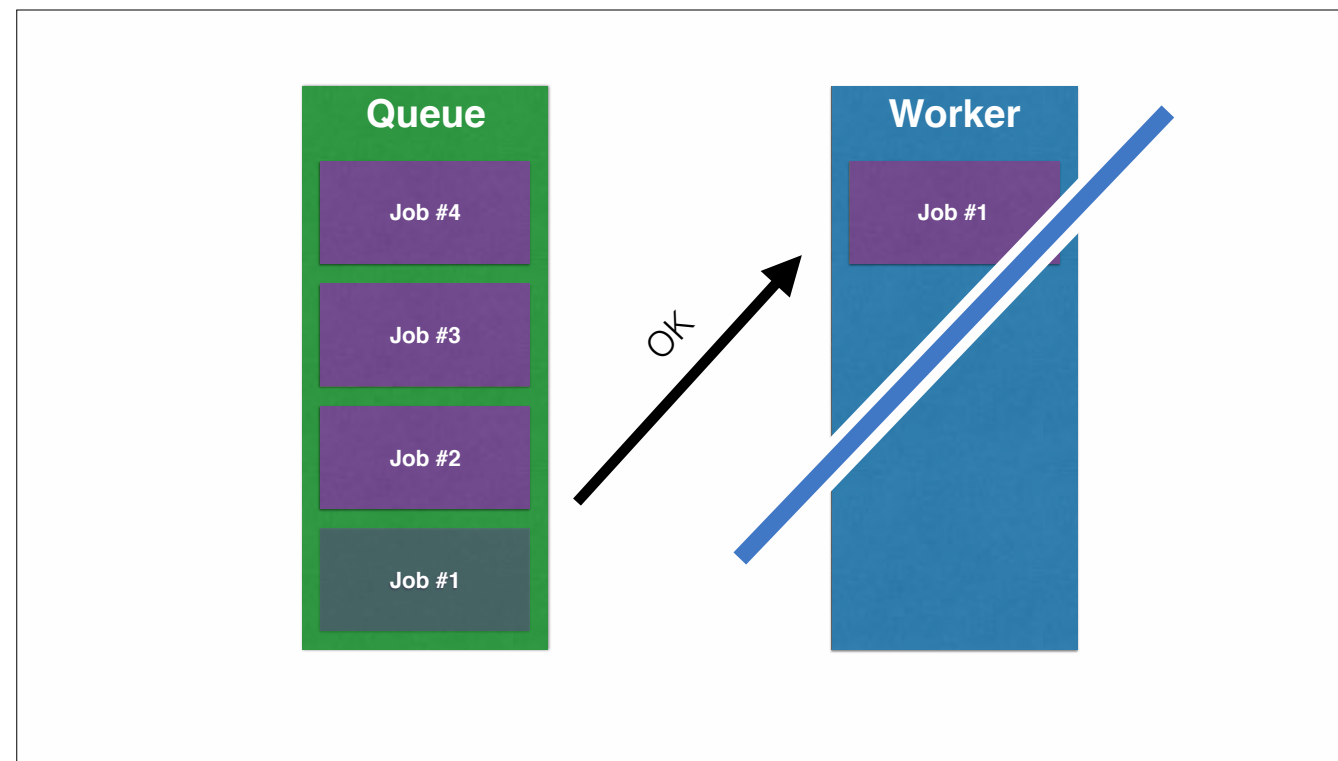
When the worker is finished it sends an ACK signal to the queue to indicate it has finished



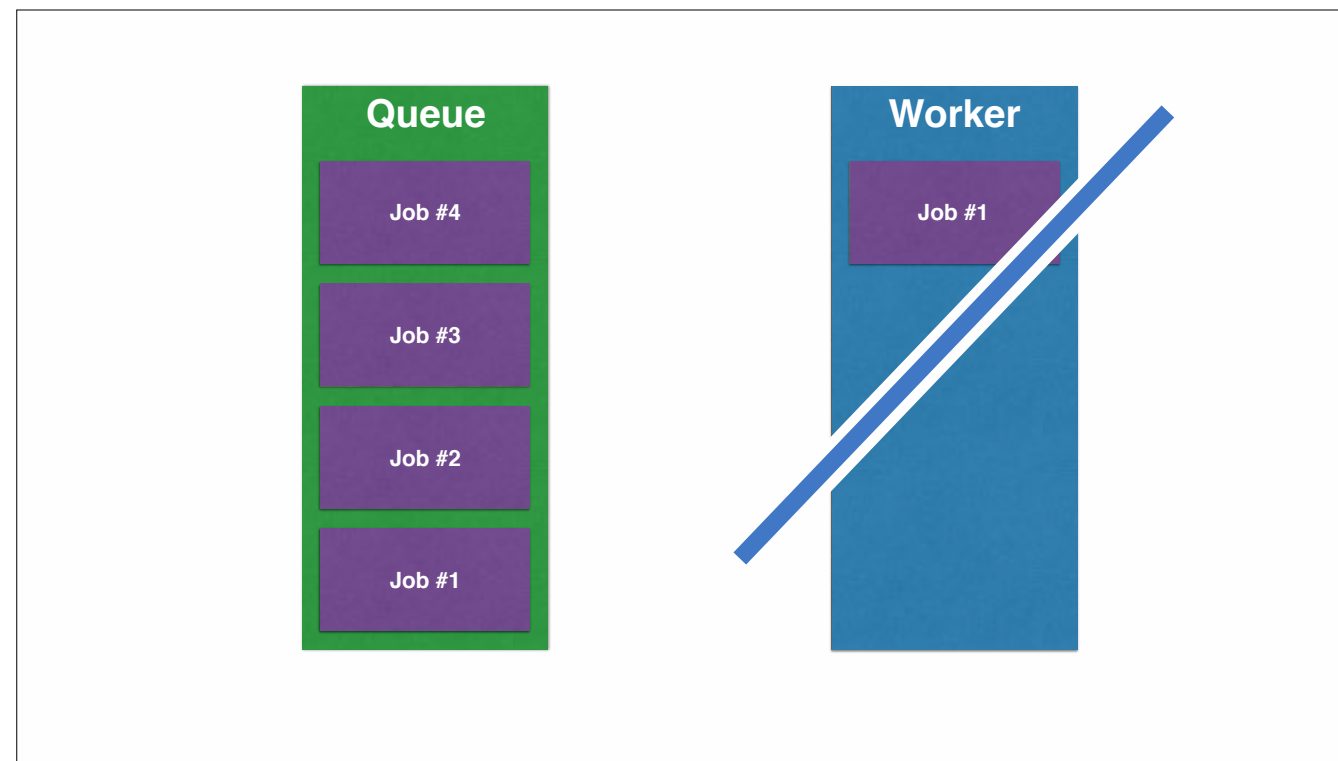
At that point the job is removed from the queue



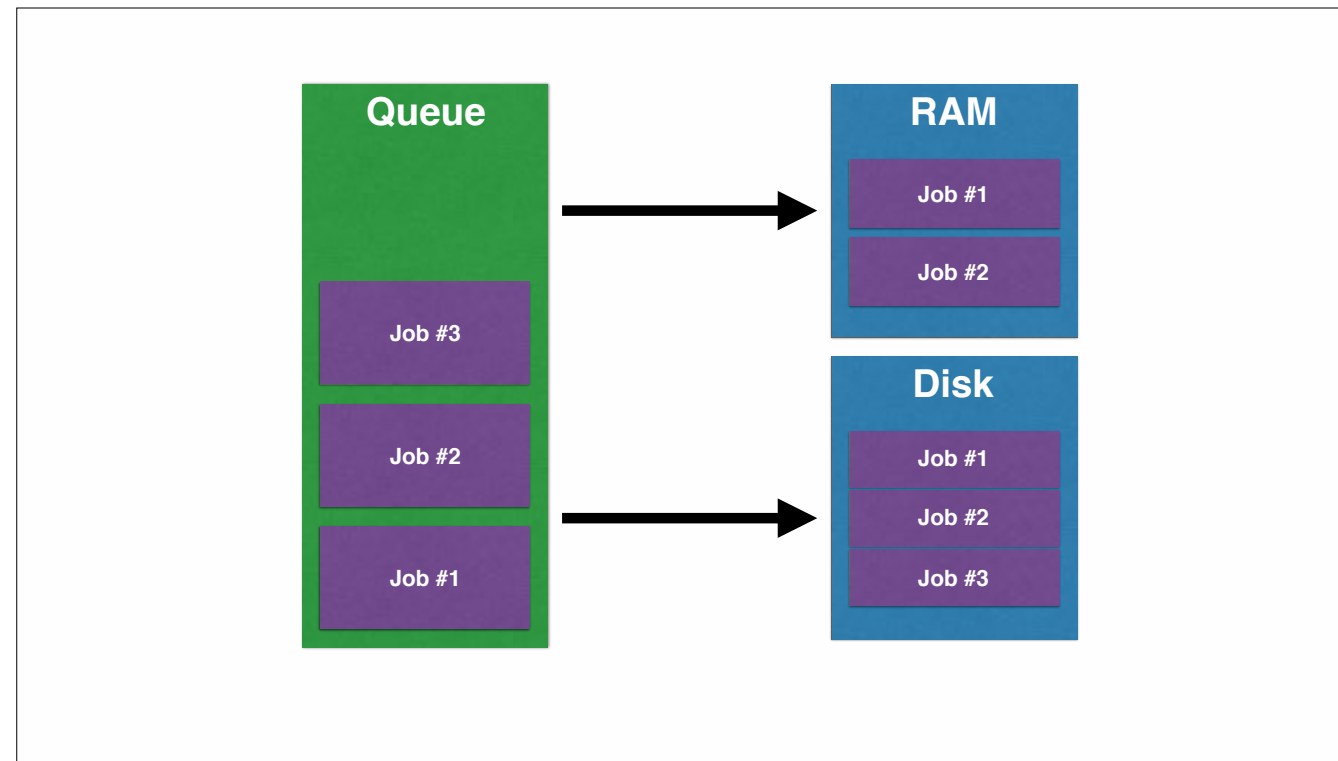
If a worker set an unrealistic goal, like in this case 1ns



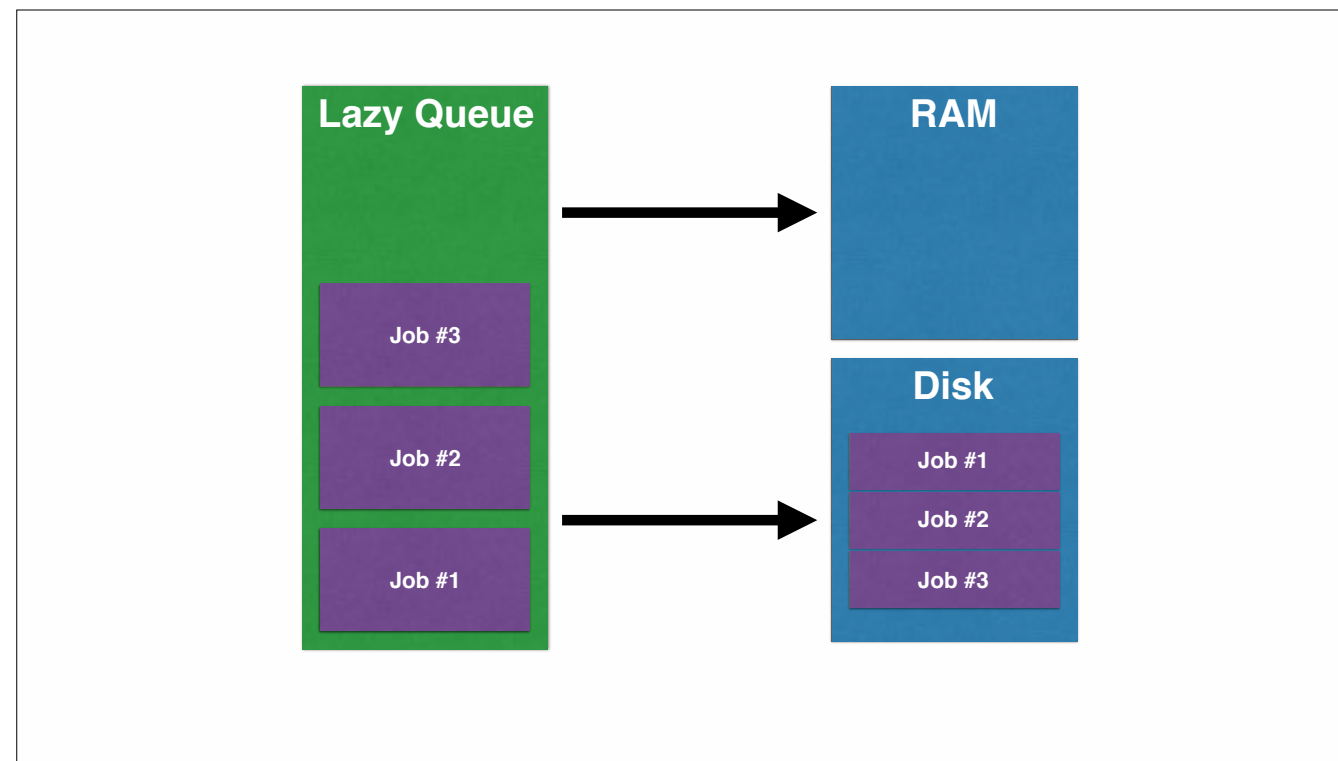
Or if a worker fails while processing



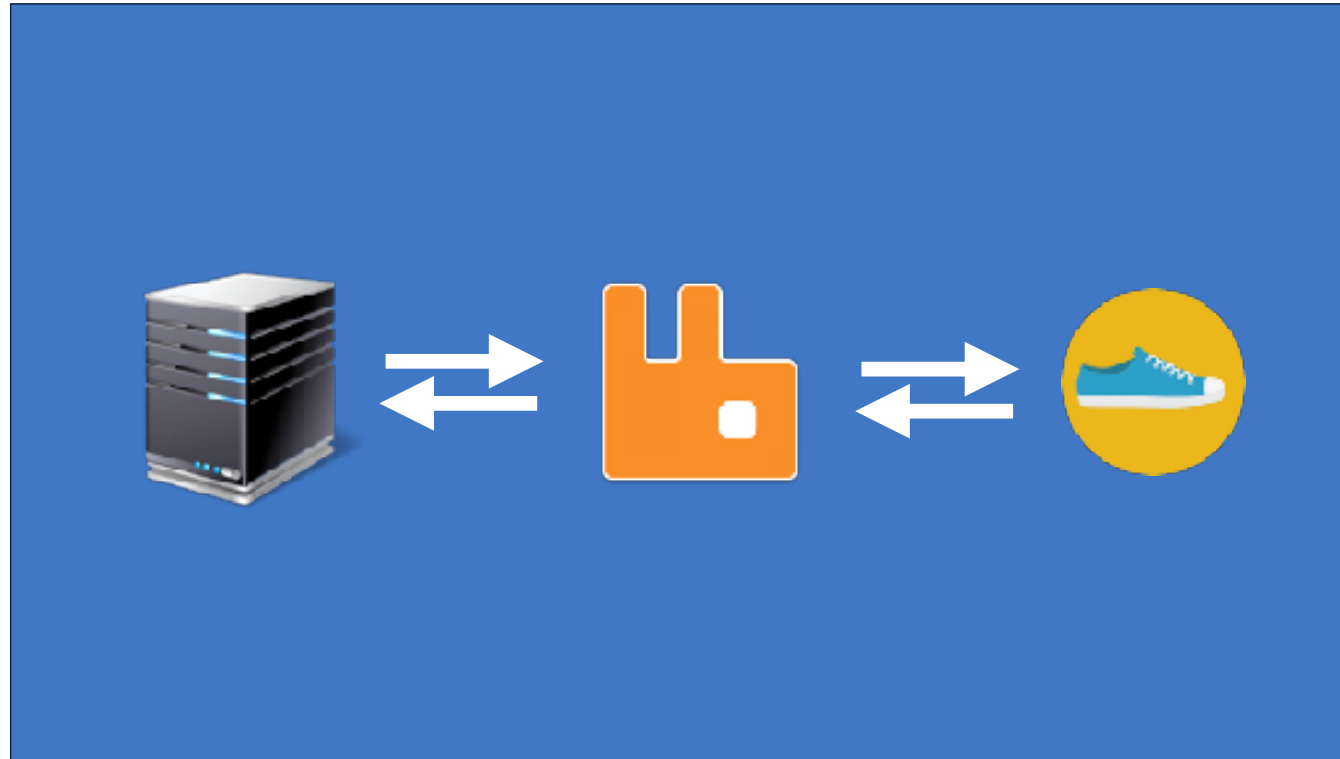
As soon as the timer runs out, the job is put back in the queue. No data is lost! Even if RabbitMQ restarts!



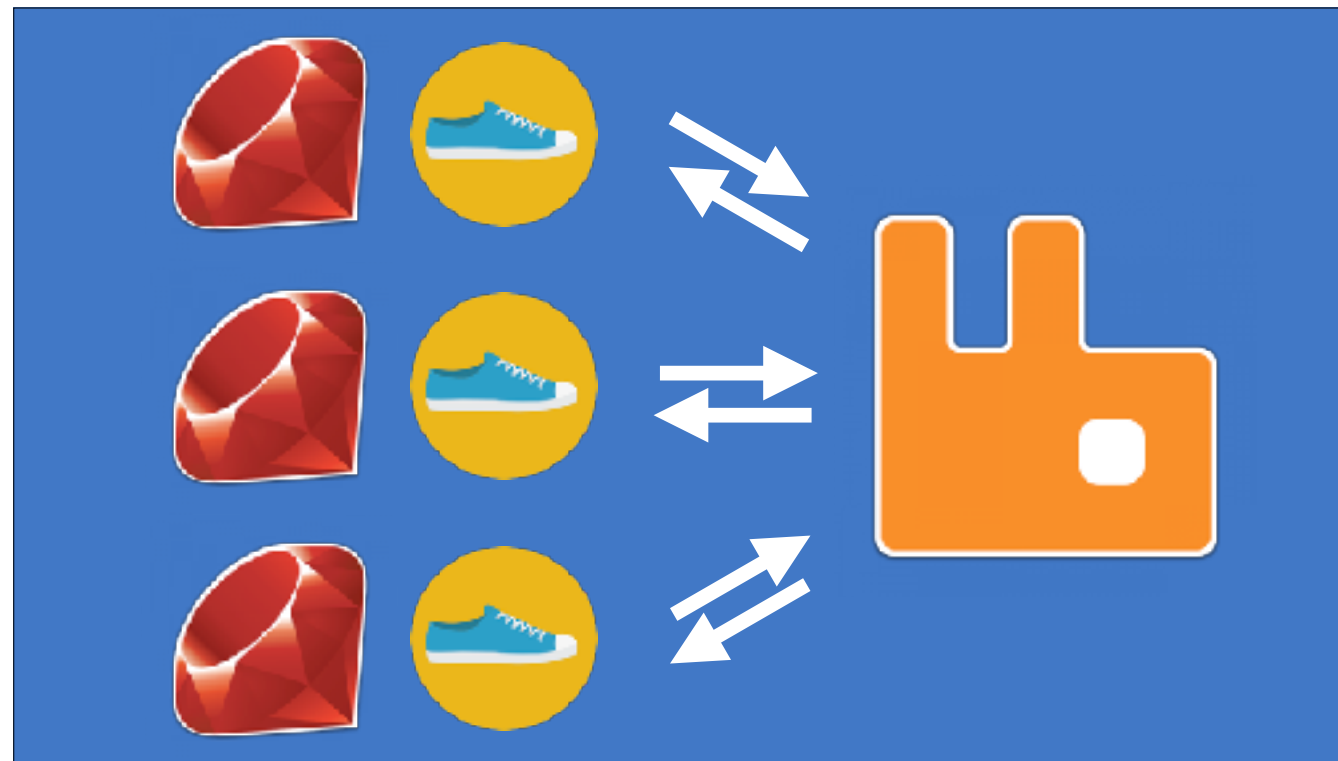
RabbitMQ always persists data to the disk in a transactional fashion, similar to Postgres. And it keeps as many jobs as it can in memory. If under memory pressure, it will start utilising the disk more and more.



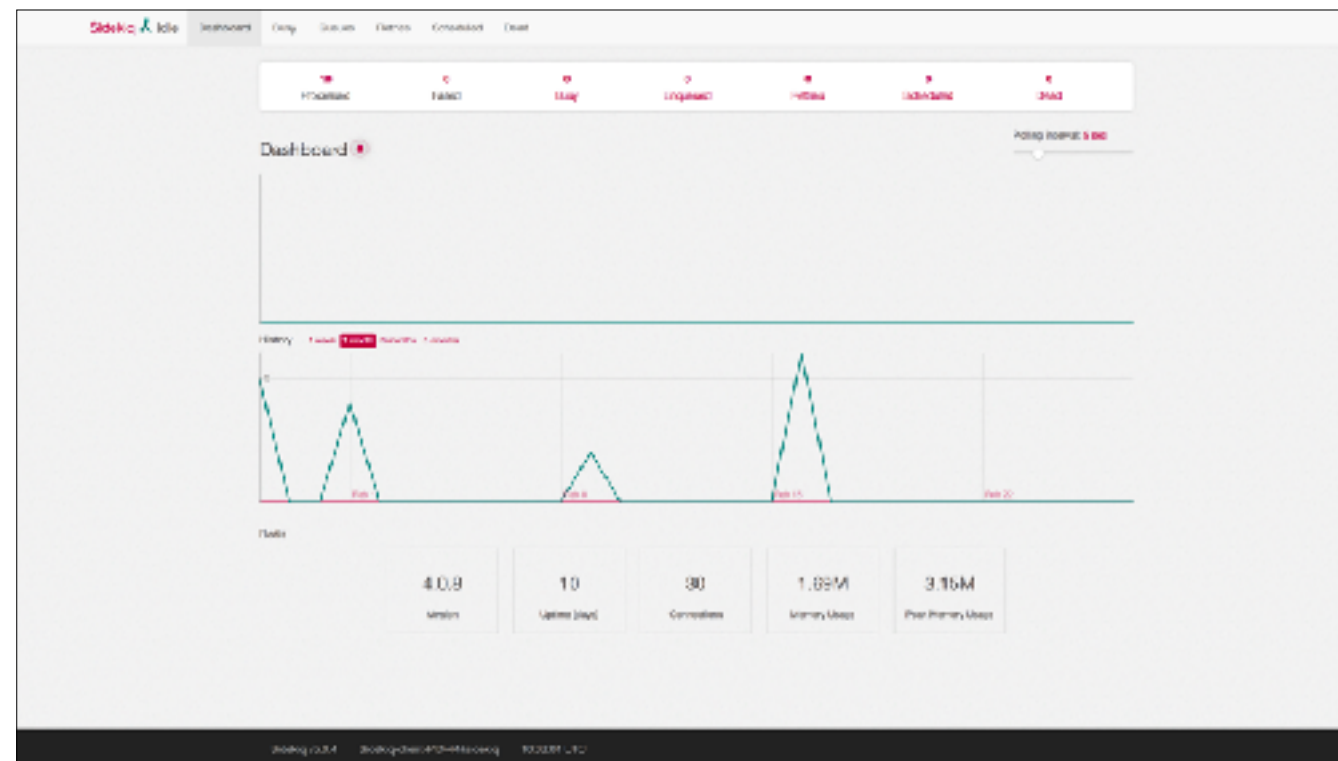
There are also other kinds of queues, like Lazy Queues, which keep everything on disk. They are intended to be used with large payloads.



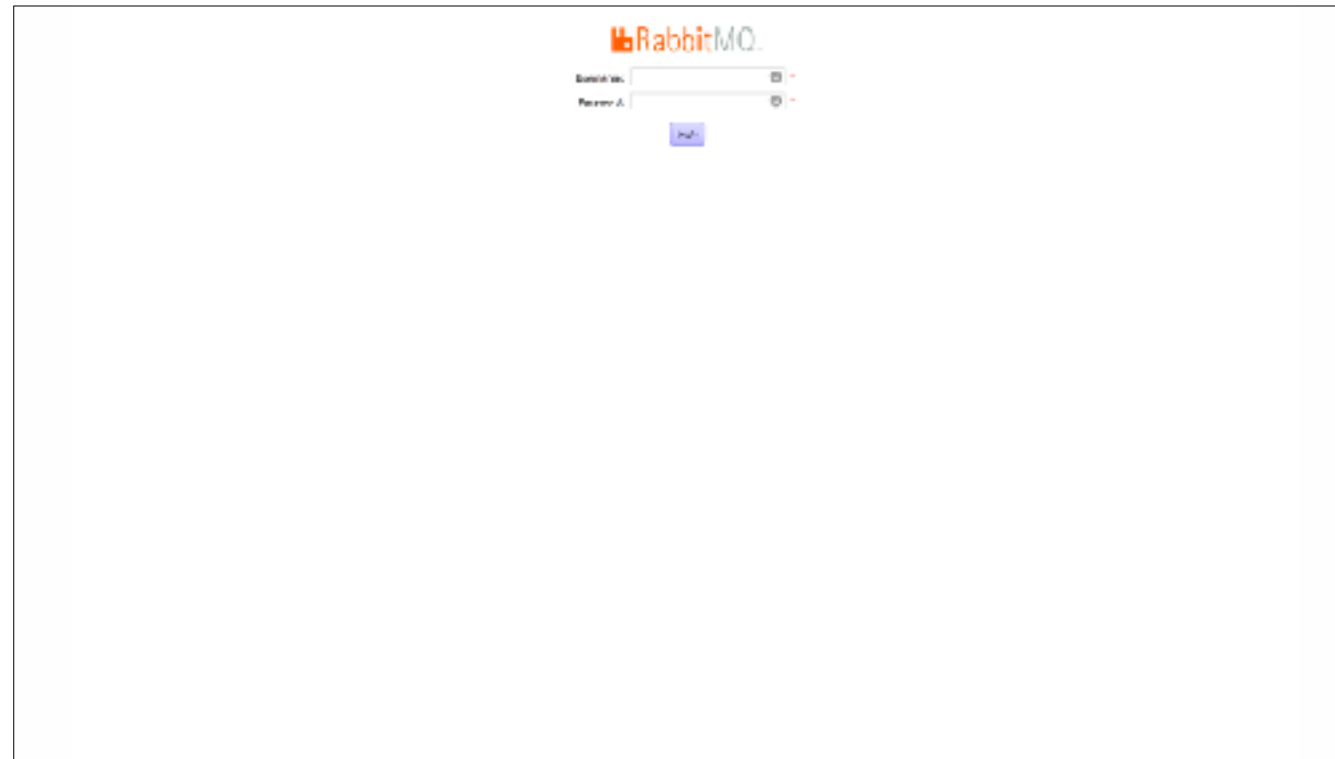
Back to our example. Since we are now able to transport basically arbitrary sizes of data through our queue we don't need access to the database anymore, and we don't need to ask our main app for data it can just give us all the information we need to process a job.






And since now we can write our workers as lightweight Ruby processes, memory consumption also goes down, and we can easily scale those processes!



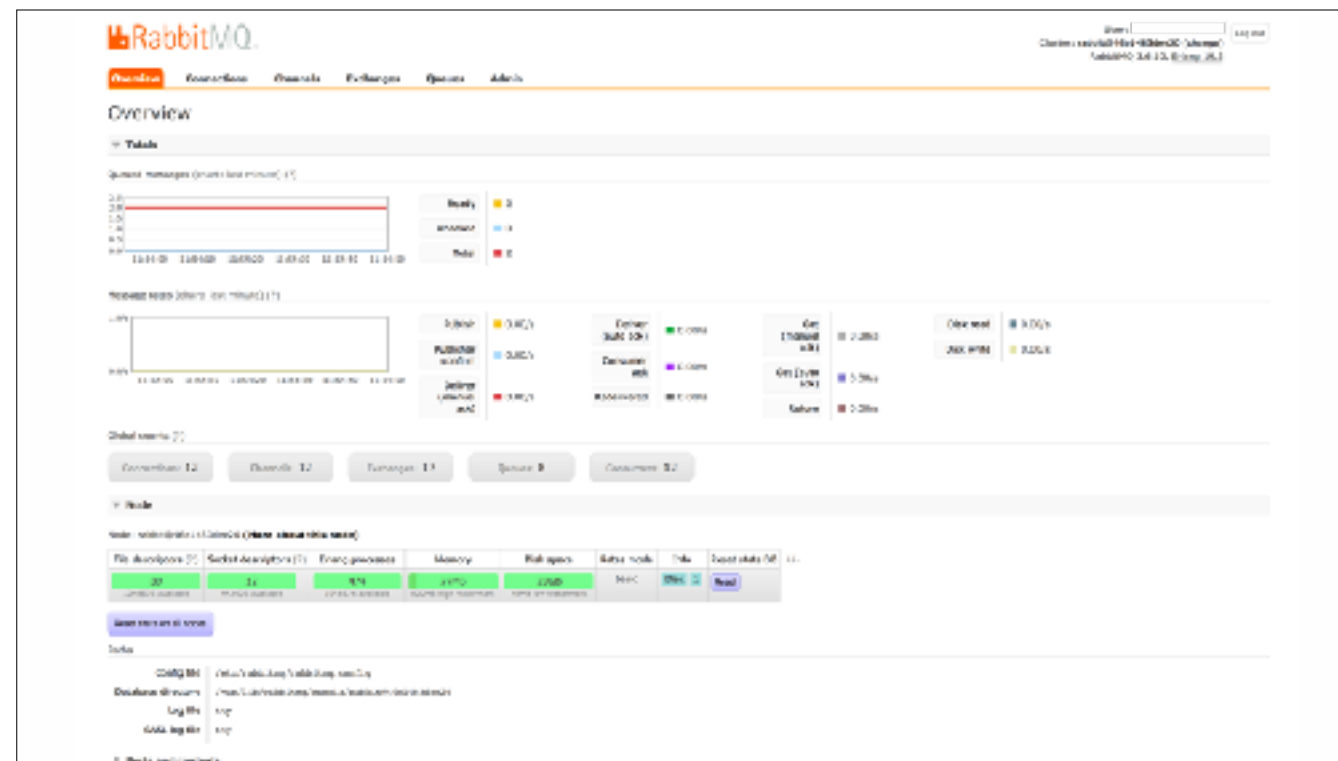
To be honest Sidekiq comes with a cool interface to manage and track how your jobs progress.



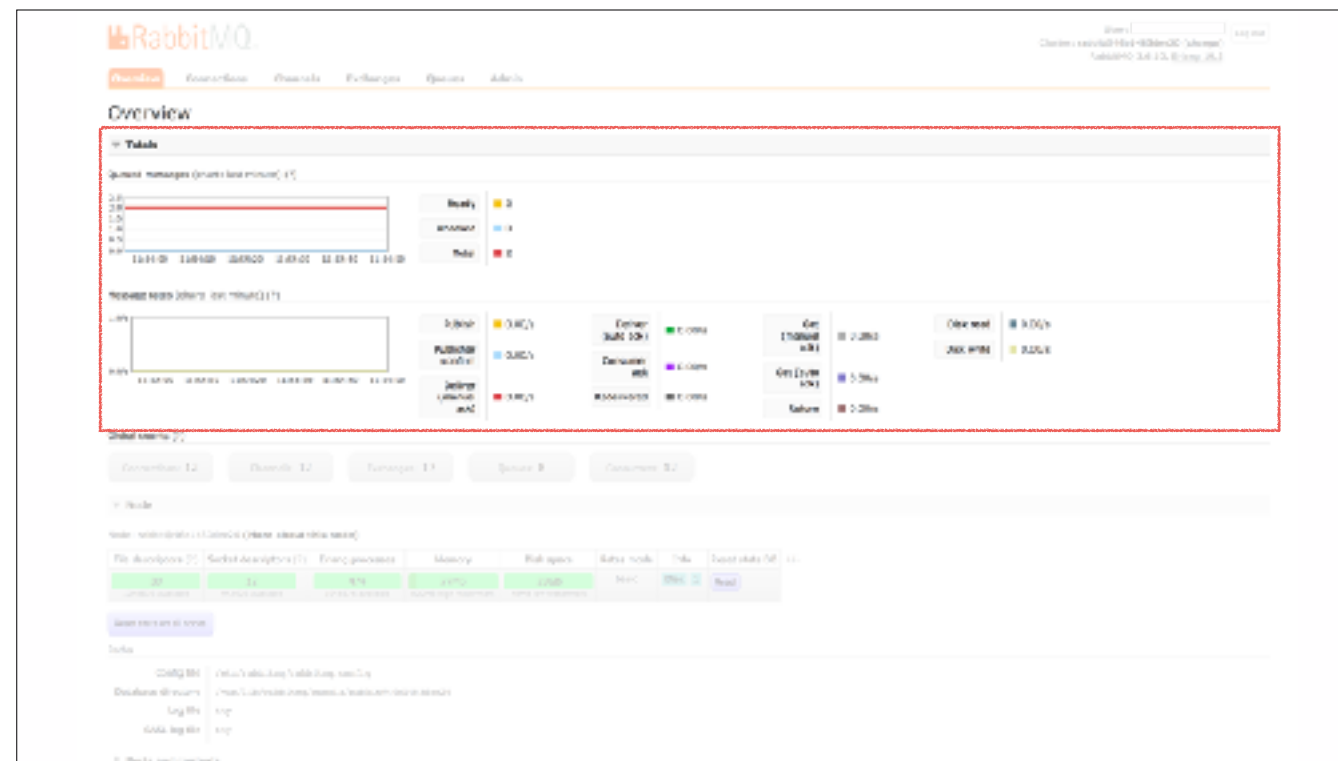
RabbitMQ also has a nice UI. In my opinion it's biggest feature is that it's free.

WORKLOAD PROJECTION			
			
DEDICATED SUPPORT	Name	Email	Email
LICENSE	UCPL	Commercial No Custom Terms	Commercial with Custom Terms
PRICING	Free	\$950 / year Unlimited Logs	\$1950 / year per 100 Workers
REQUIREMENTS	n/a	Credit Card	Credit Card, Invoice

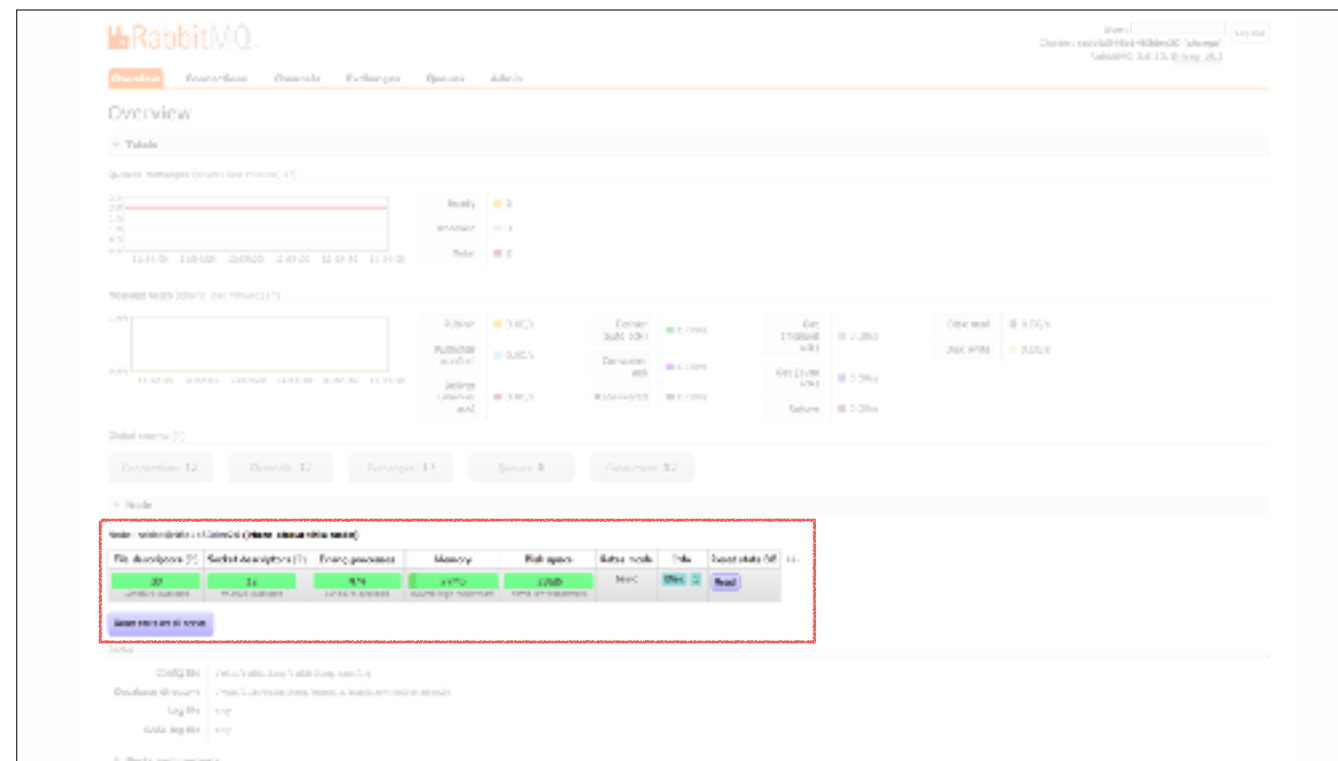
And it comes with authorisation, that's a hell of a lot better than \$2k for a simple login screen.



Anyway, you have the standard overview screen.



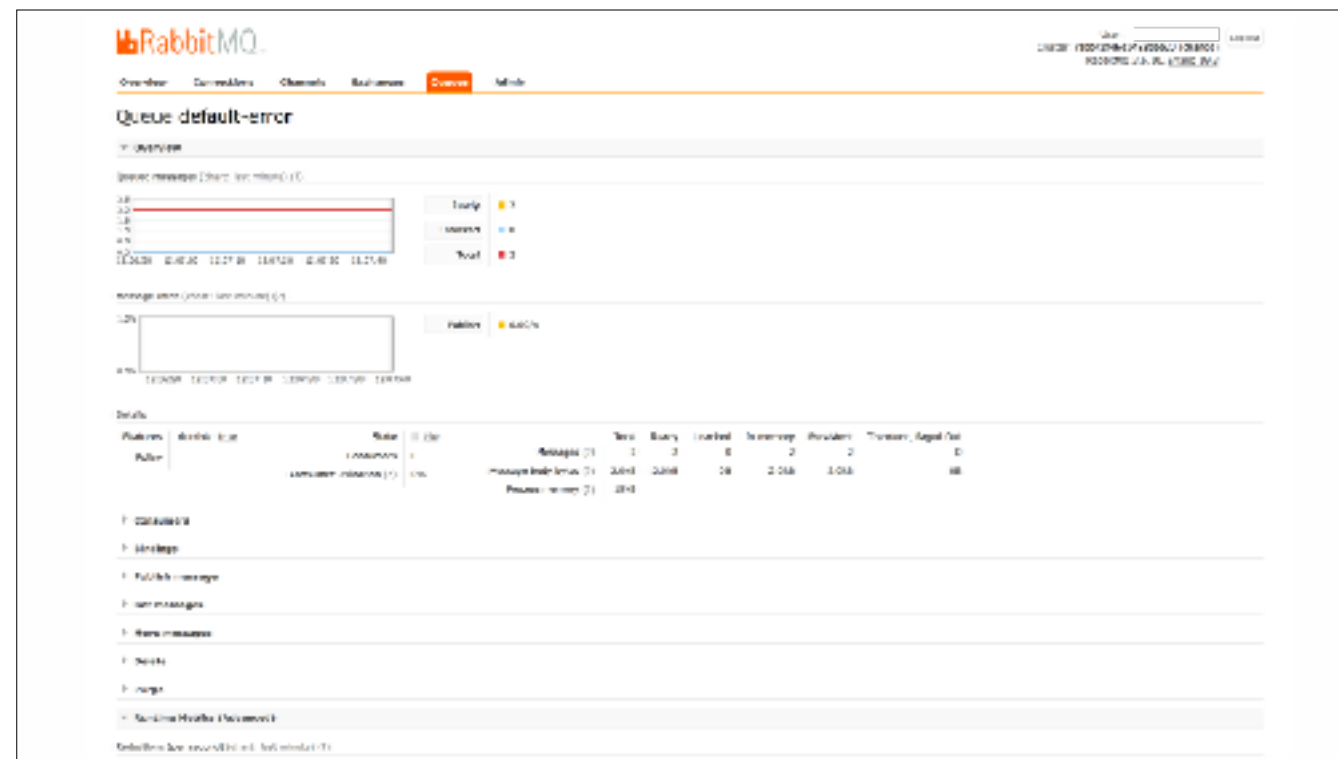
It show how many messages are being processed.



And how the systems resources are being utilized.

[illegible]

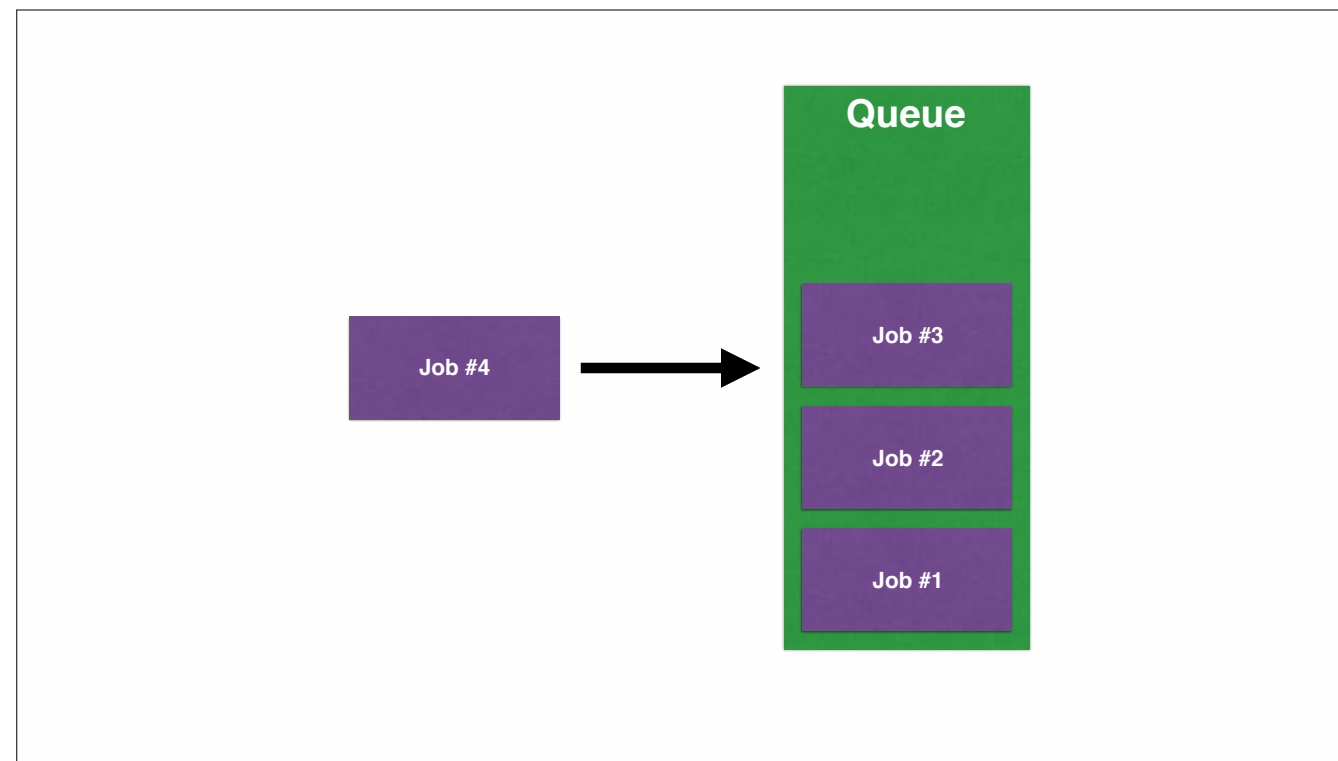
You can also manage individual queues.



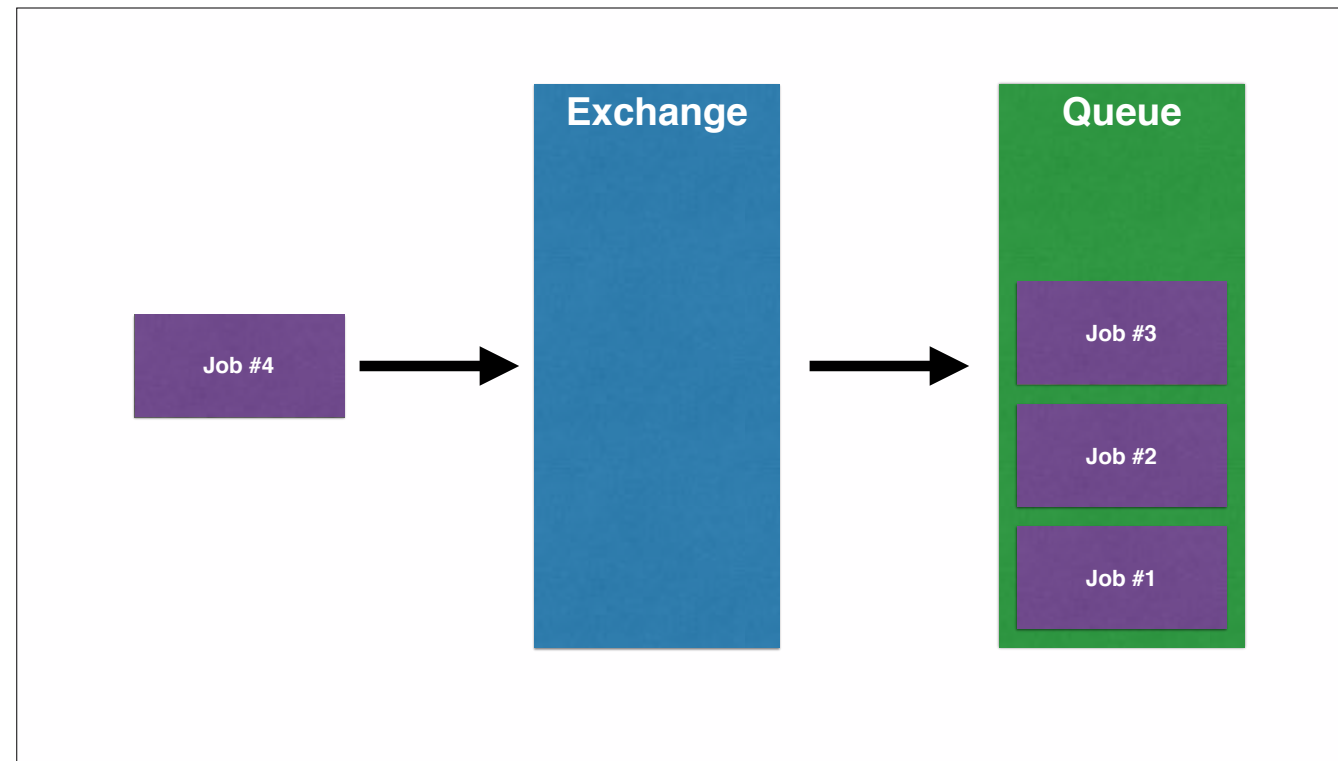
You can look at it's individual statistics. But you can also modify it, read, create and delete messages. And a lot more.



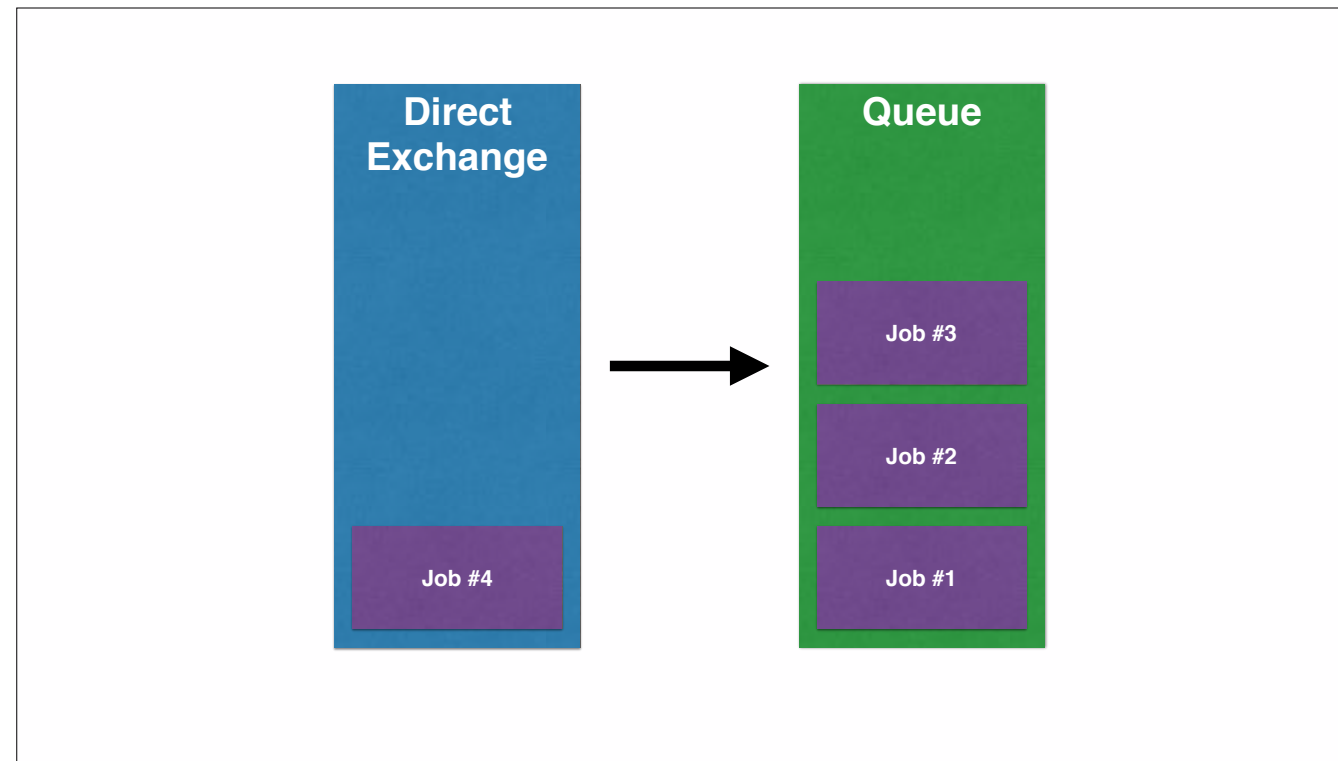
This brings us to exchanges. As they are the only Tab that most people won't know what it does.



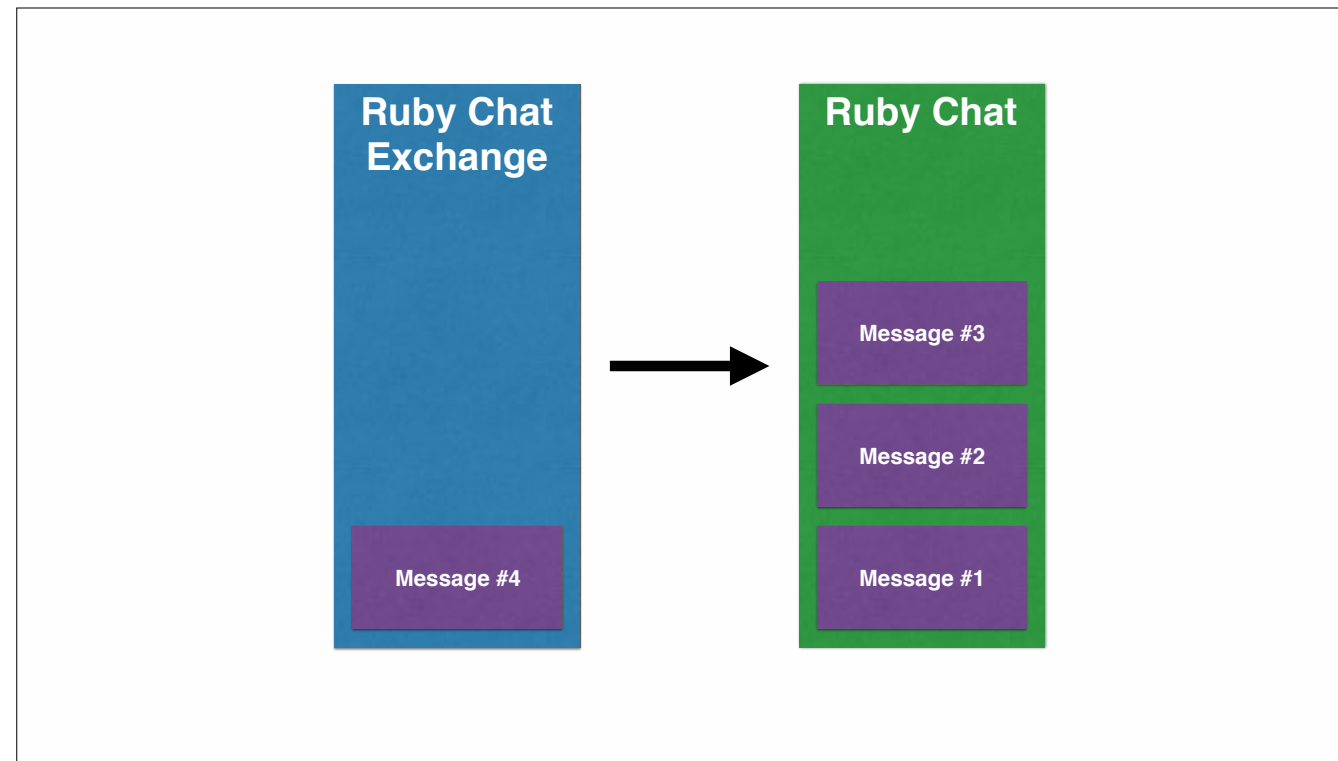
Exchanges are a feature of AMPQ. Using AMPQ it's actually impossible to push something directly to a queue.



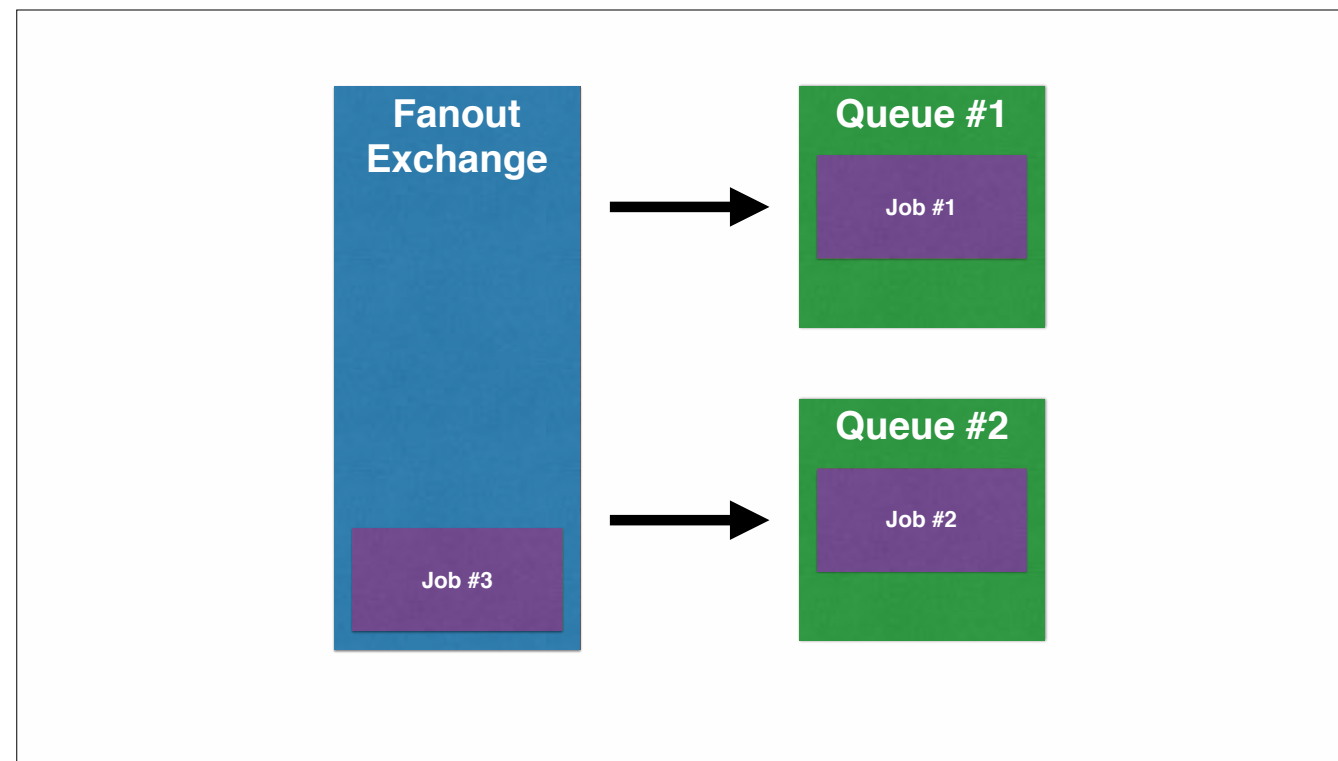
In AMQP, it's only possible to push a message to an exchange, and the exchange decides in which queues to put the message.



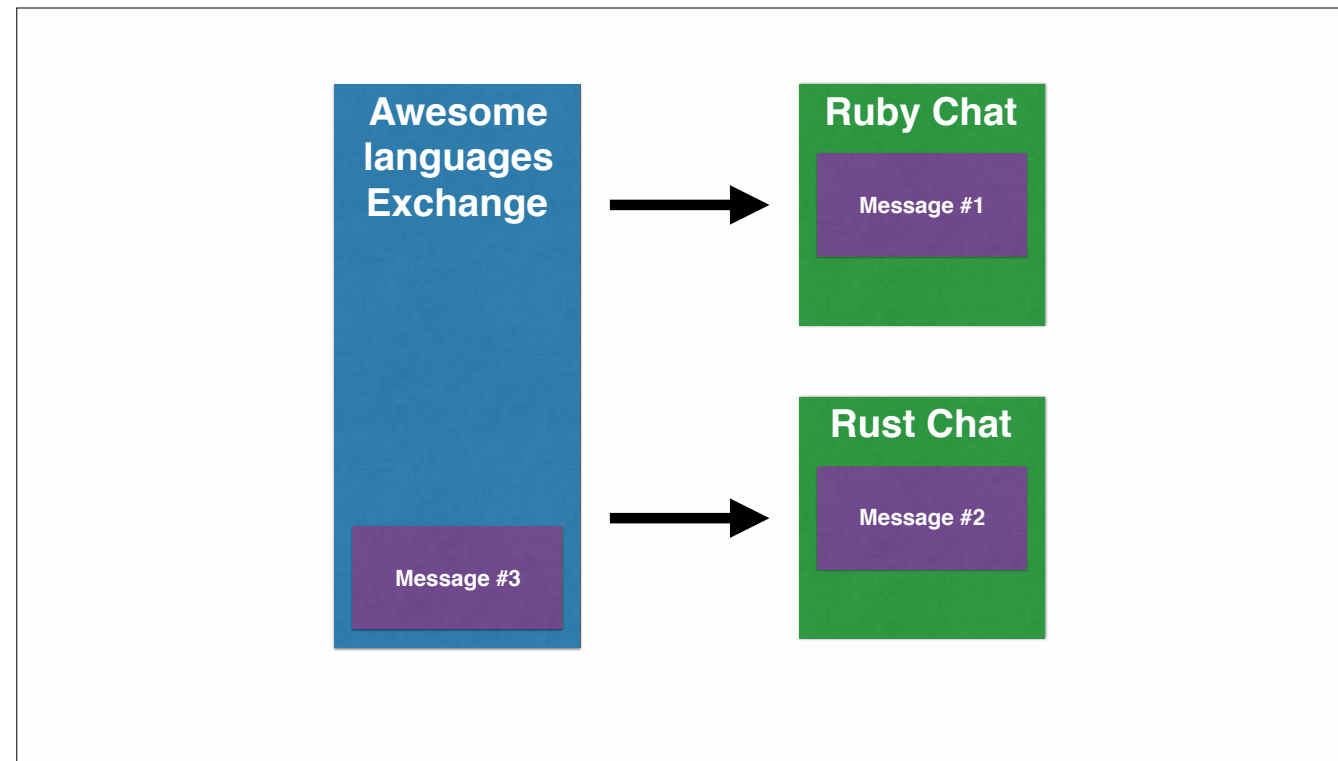
There are four types of exchanges. The above is a direct exchange. Messages passed to a direct exchange are directly passed to a single queue. This is the way that Sidekiq works. You push a job to a queue and it gets processed.



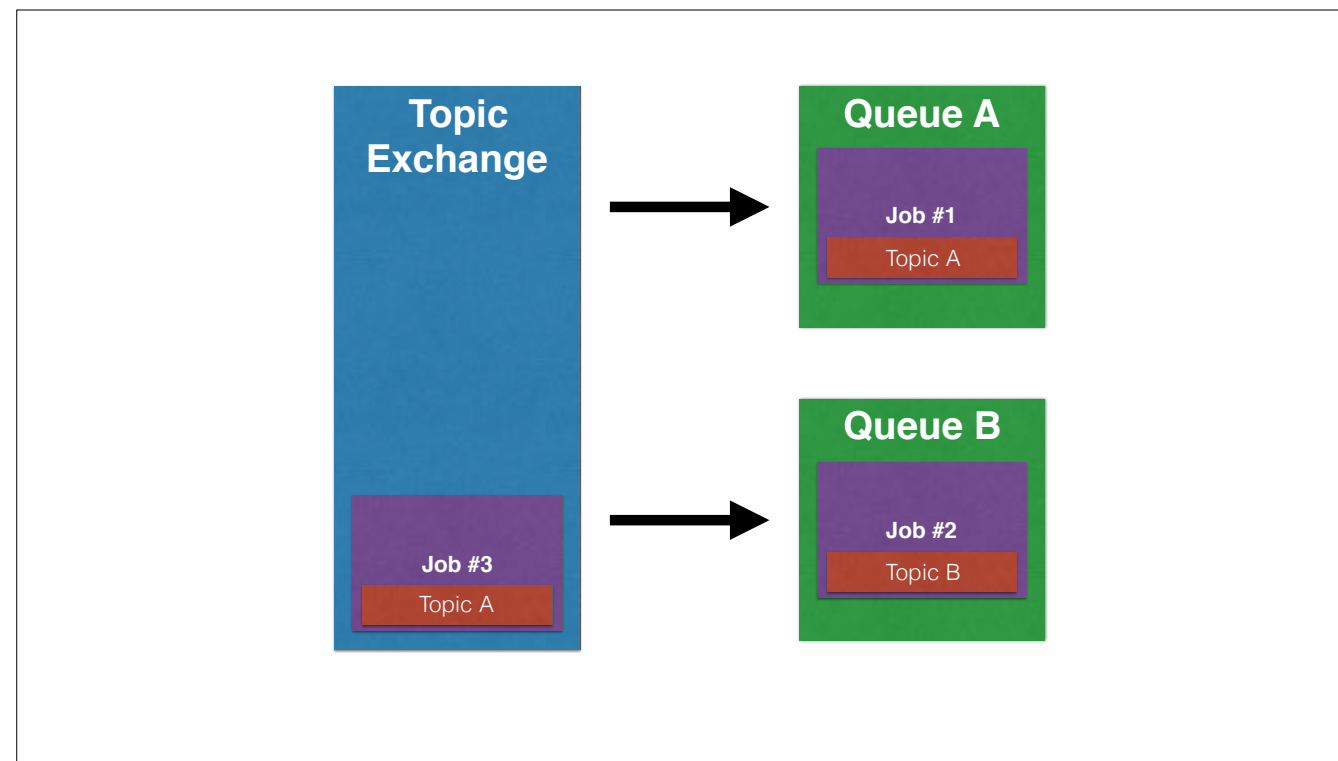
If we were building a chat, direct exchanges would be the same as sending a message to a single chatroom. In this case Message #4 would be delivery to the Ruby Chat, and only to the Ruby chat.



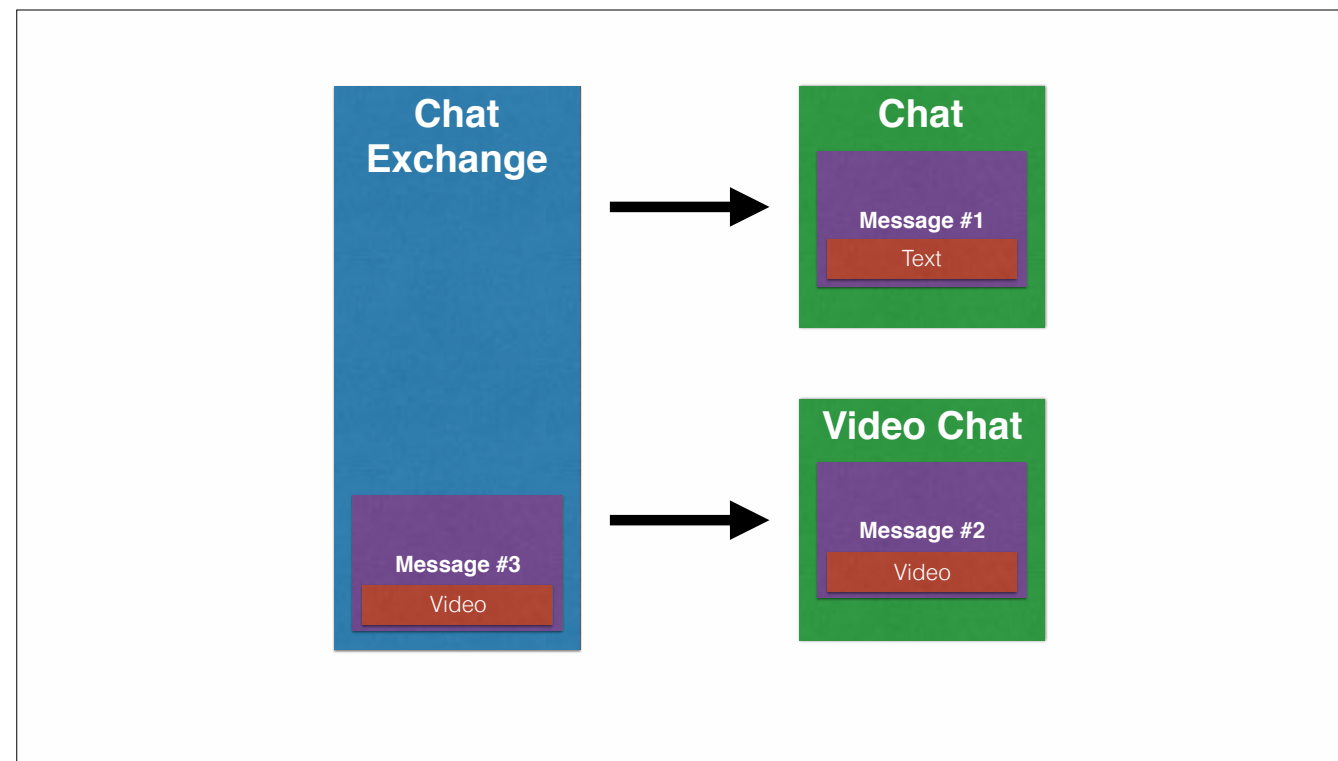
Then there are fanout exchanges. They send out the message to all bound queues.



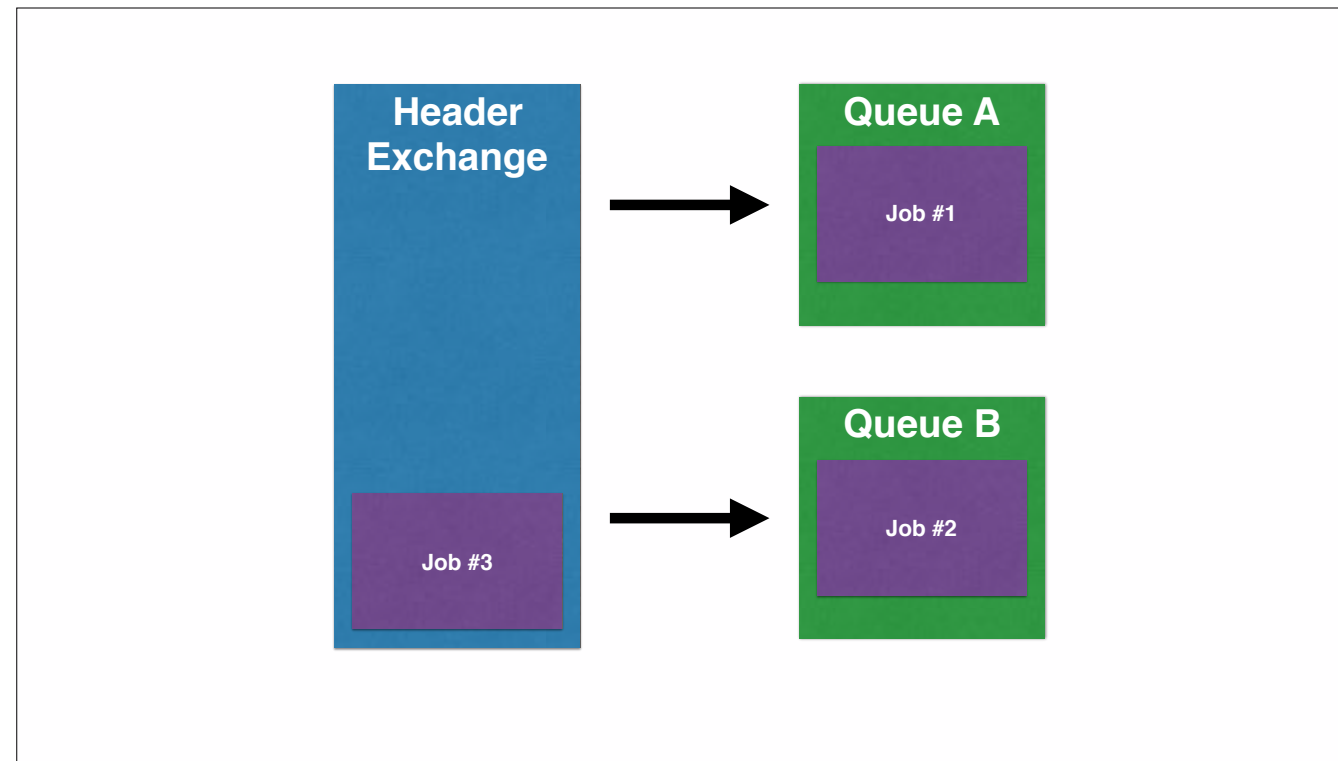
A practical example for this is a broadcast in a chat room. Let's say we want to broadcast a message to all chatrooms of awesome languages. This is a good use case for a fanout exchange. In the above case, Message #3 will be delivered both to the Ruby Chat queue and the Rust chat queue.



Topic exchanges are simple routers. With each message you publish you can specify a topic. The exchange will look at that topic and send the message to all queues bound to that topic.



In our chat example, you would use a topic exchange to direct messages to specific pipelines. E.g. if your users can upload images and video, and you want to process them before sending them in the chat. Video messages would get tagged and directed to a queue that processes them first, and then sends them out.



Lastly, there are header exchanges. They are the same as topic exchanges but more powerful, they can use nearly any part of the message to determine where to route the message. The sample for this is the same as for topic exchanges, therefore I'll skip it.



But what would you use the?

Why would you use exchanges if you can do all that logic in plain Ruby?

The first reason is speed. Doing this kind of routing is quite slow with Ruby.

The second exactly-once-delivery, if your Ruby process restarts it will get to send out all the messages again.

The third reason is a pragmatic one. That logic is not in your codebase. In other words, you don't have to maintain it.

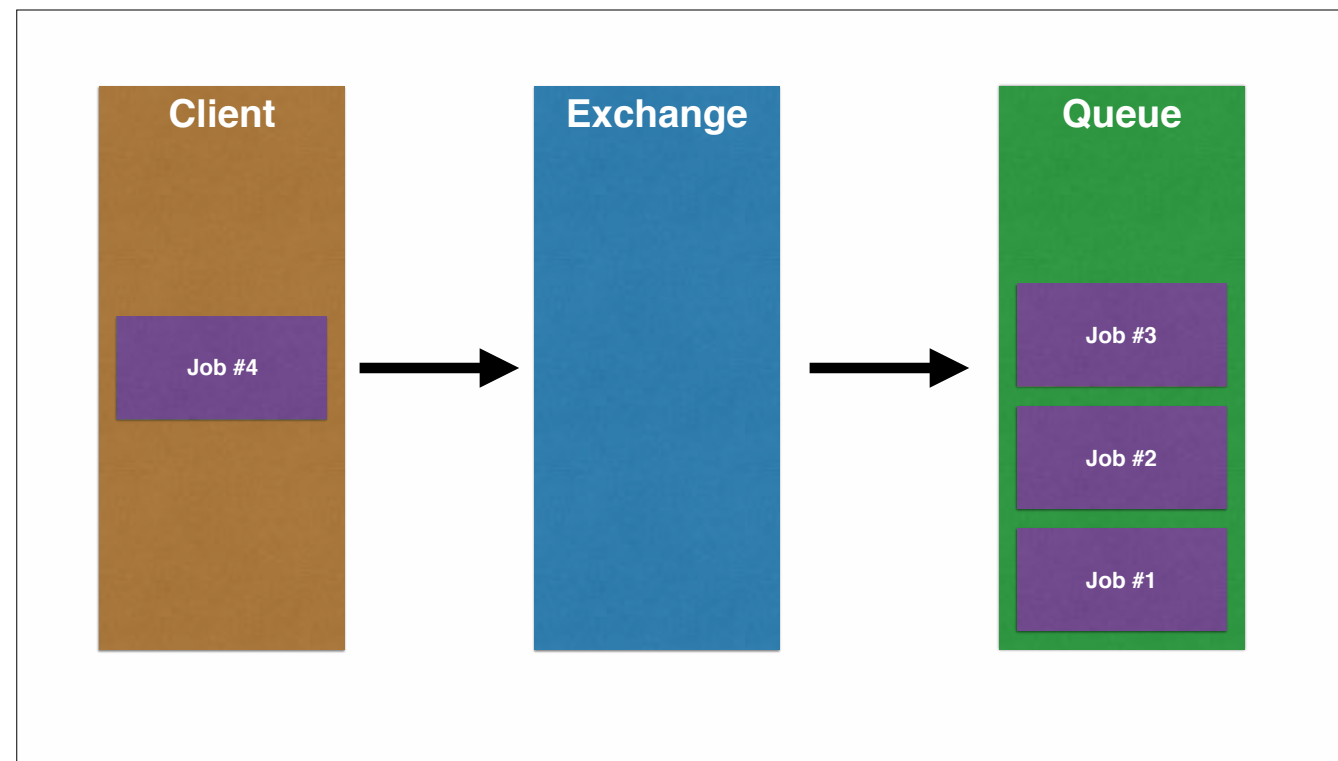
The final reason is deprivation. With exchanges it's much easier to deprecate a service, simply point the exchange to a new service and you are done.

03 SPECIAL FEATURES

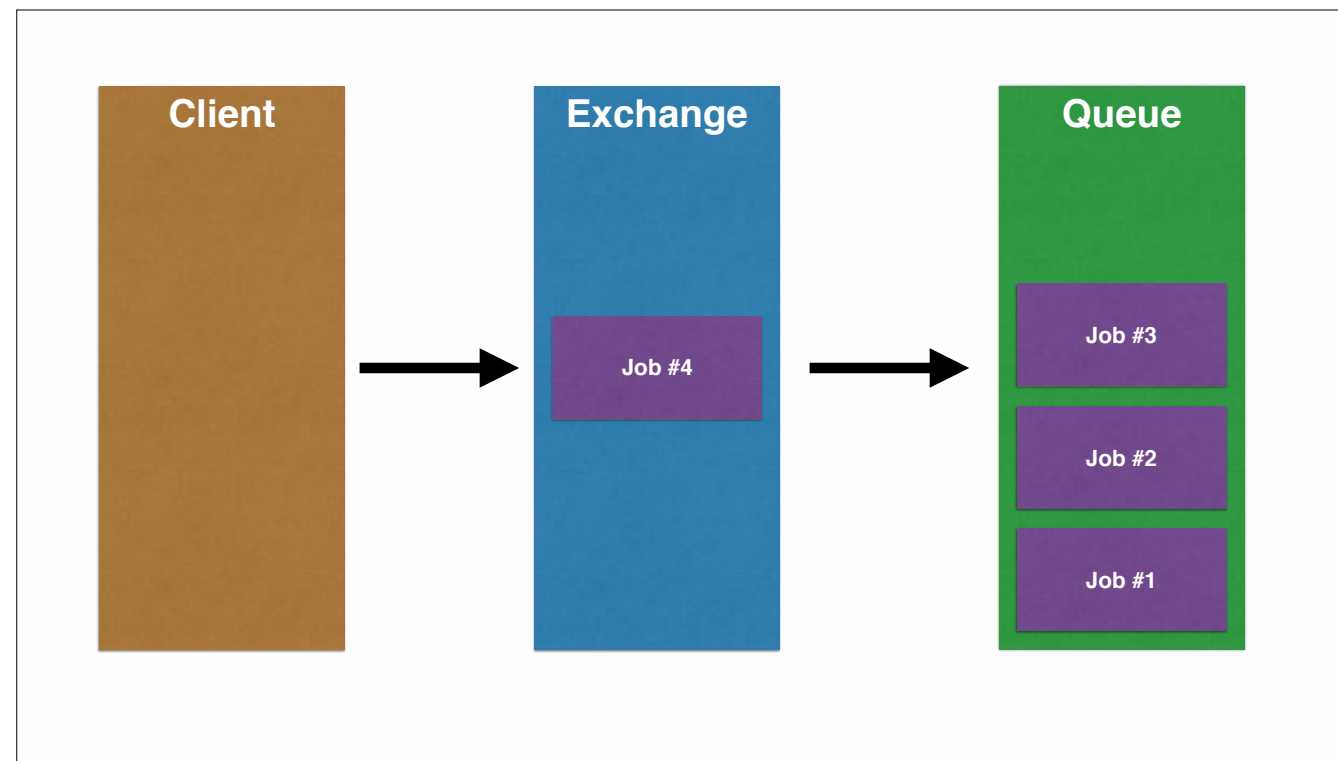
RabbitMQ comes with a couple special features non-standard for AMQP

Direct reply-to

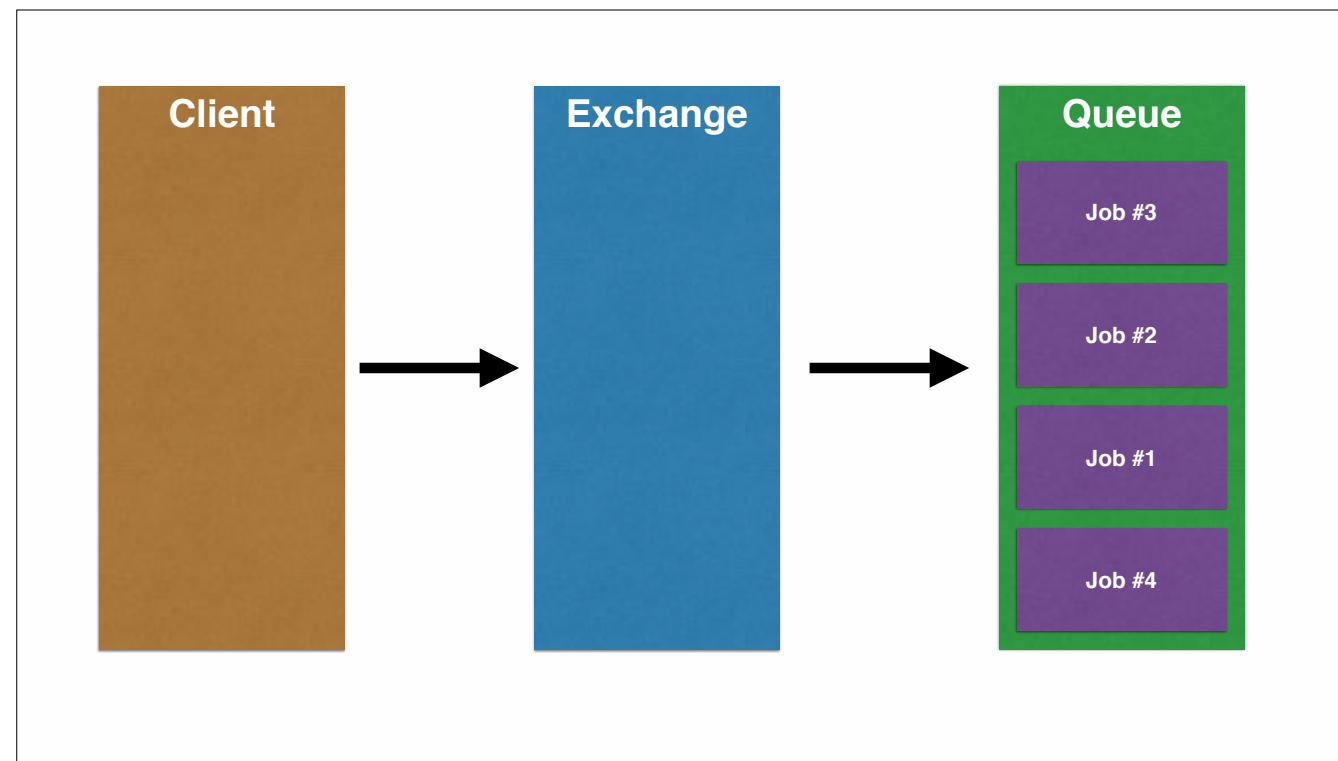
The first, and perhaps most useful feature is direct reply-to. It enables synchronous IPC.



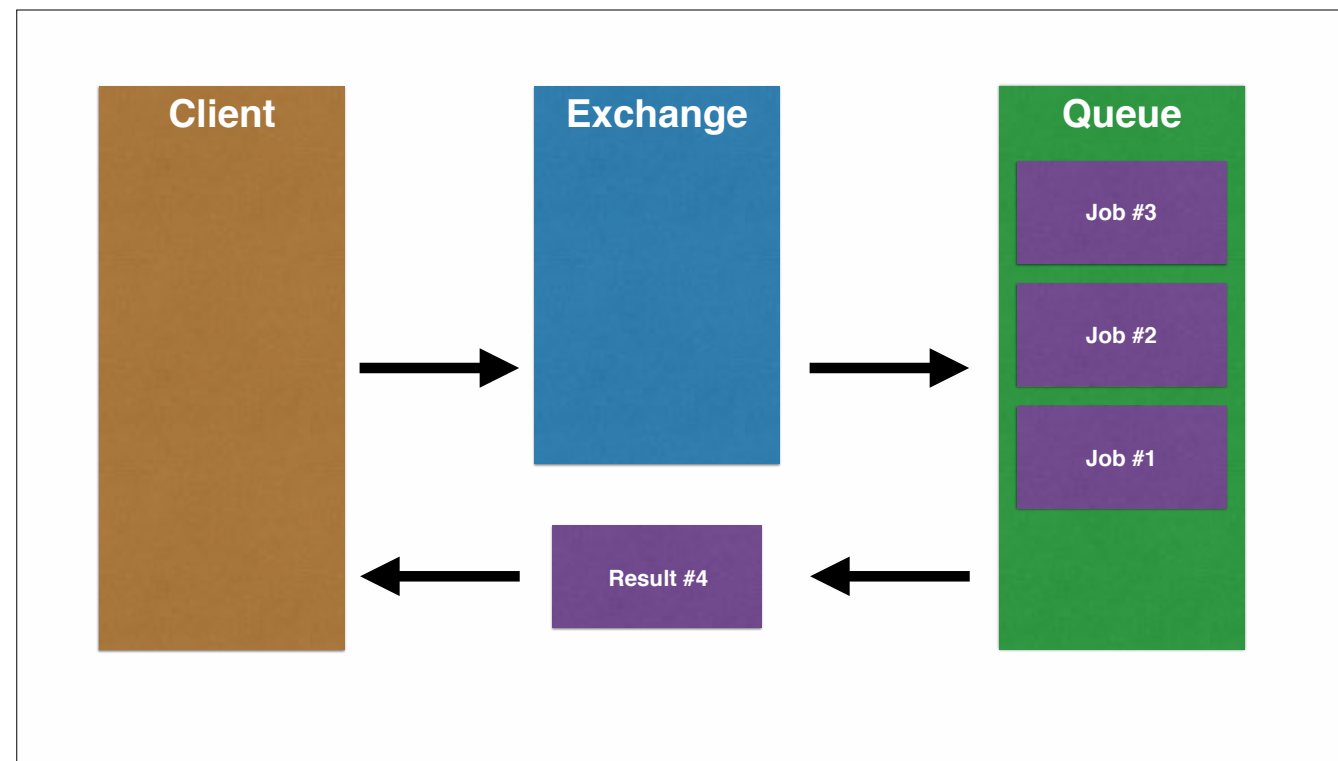
When a client sends a message



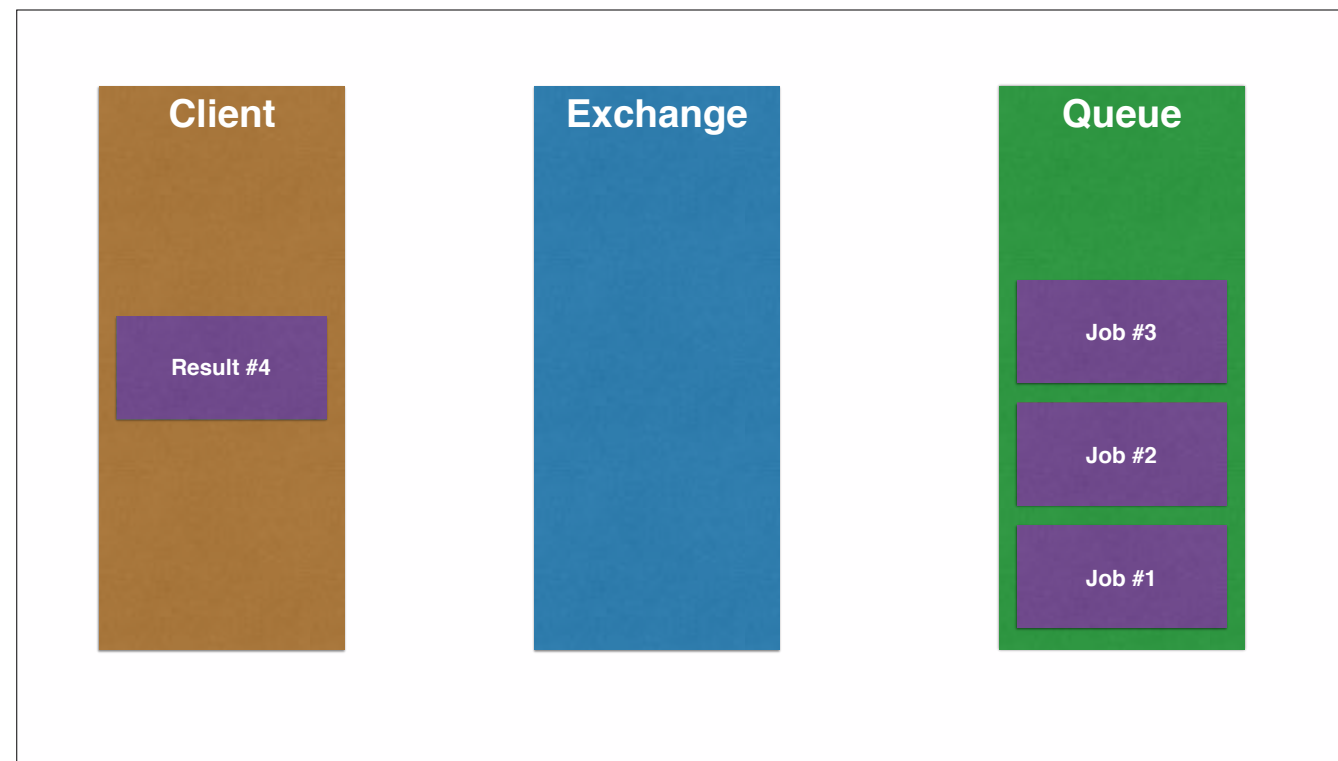
It's received by the exchange and routed to a queue



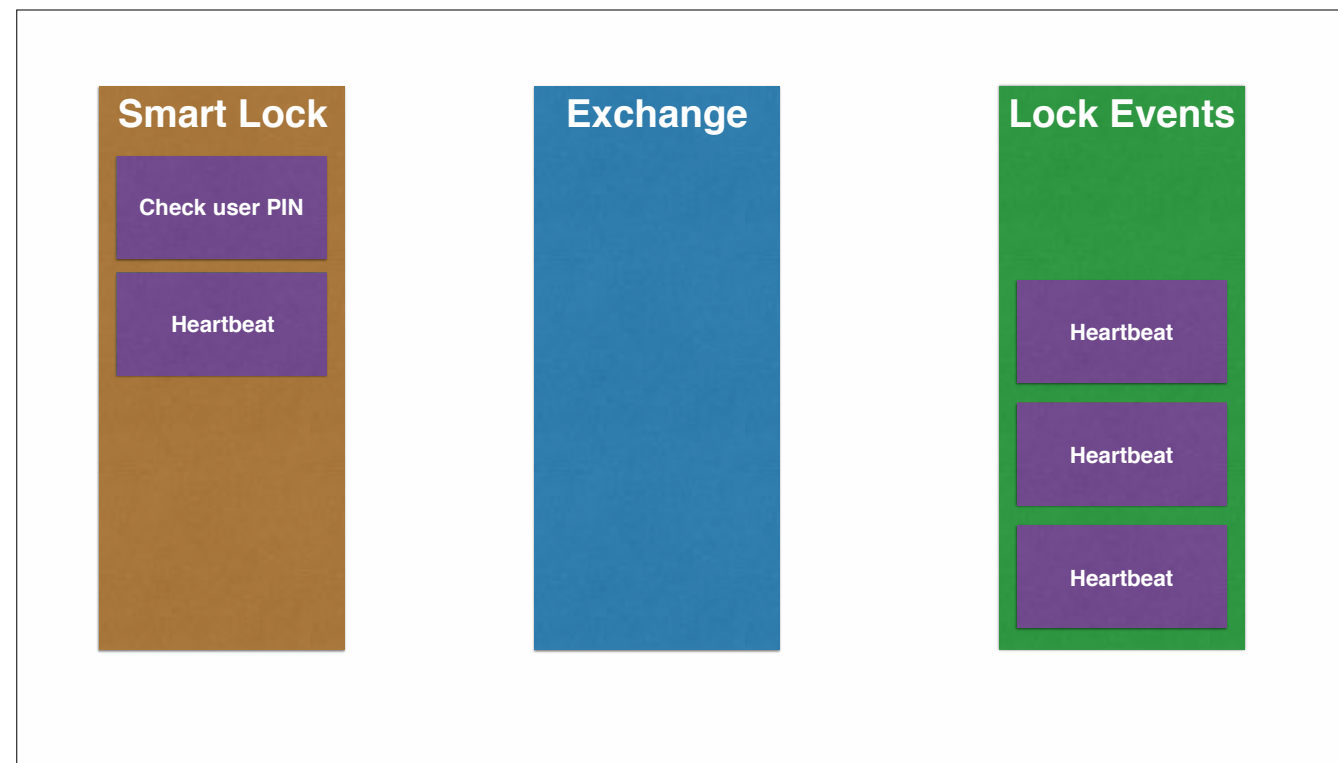
At the queue it's handled like a priority message. e.g. it gets processed first.



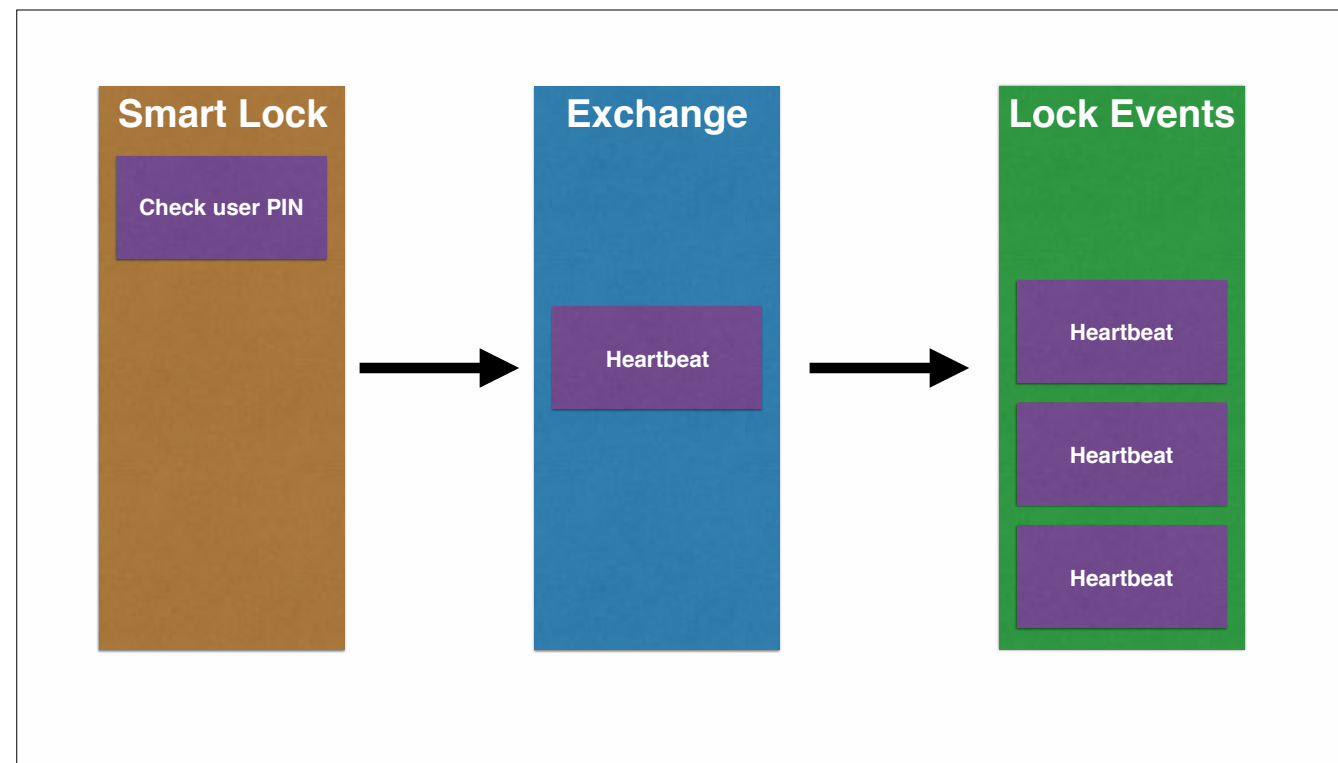
Now the special part. The result is passed back to the client! Directly!



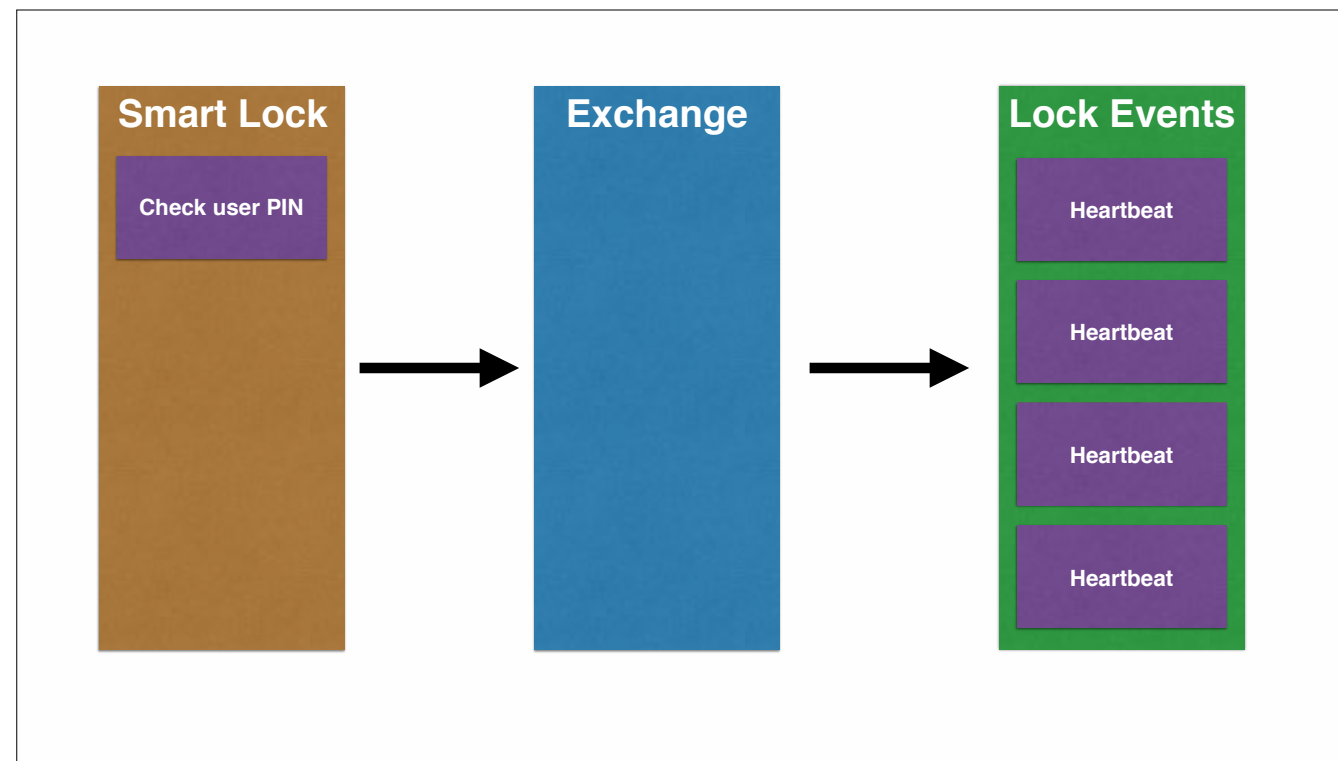
And then execution continues as normal.



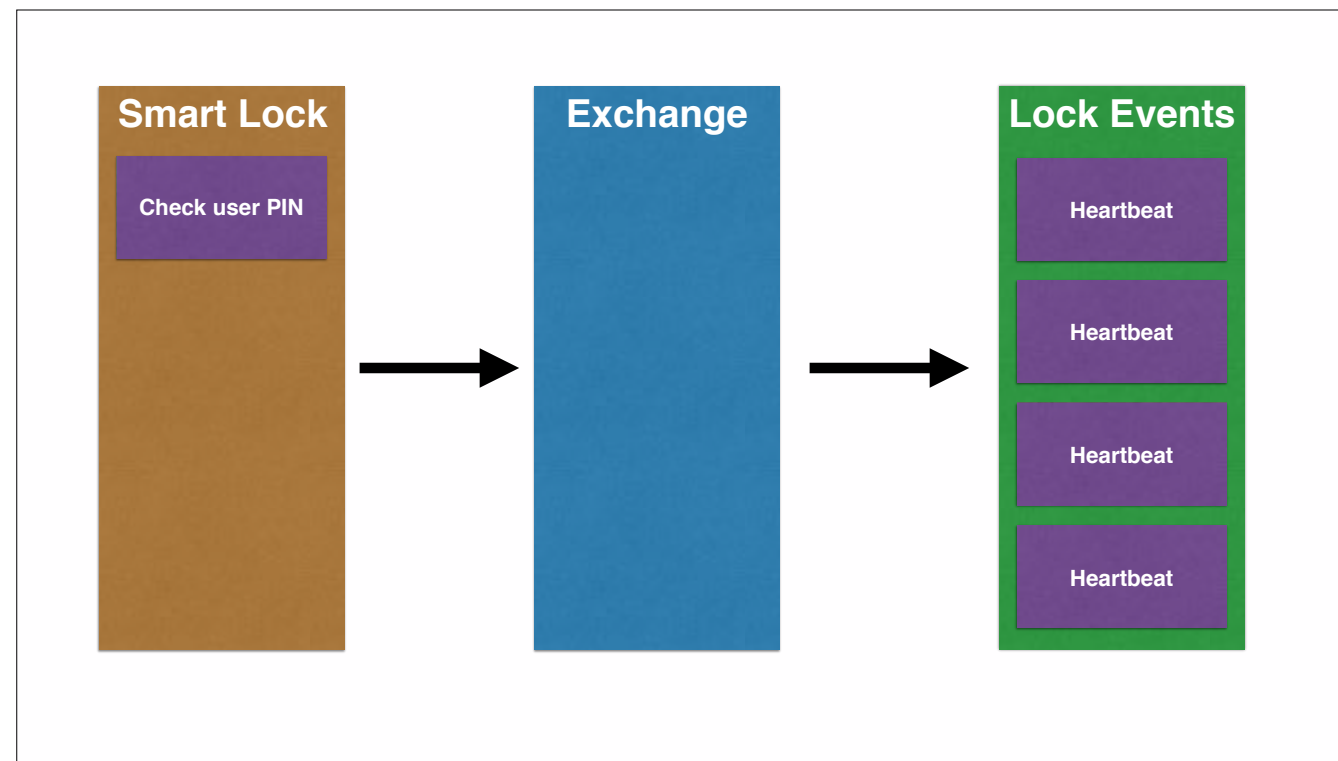
A real world example with a smart lock would be as follows. We have a smart lock, it needs to send out a regular heartbeat to indicate that it's still online and connected. But it can also check if a user's pin is correct.



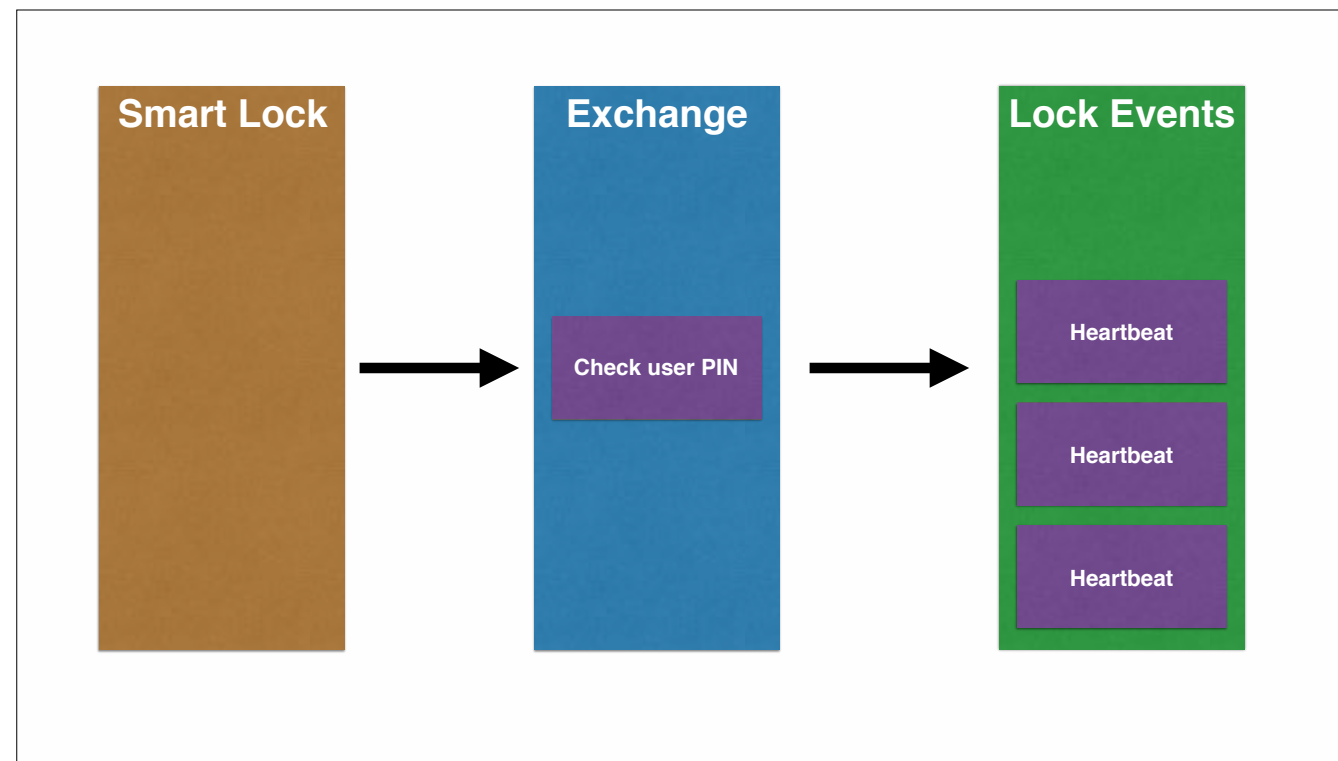
A heartbeat is not that important of a event that we need to stop the world and process it, so we put it into the queue



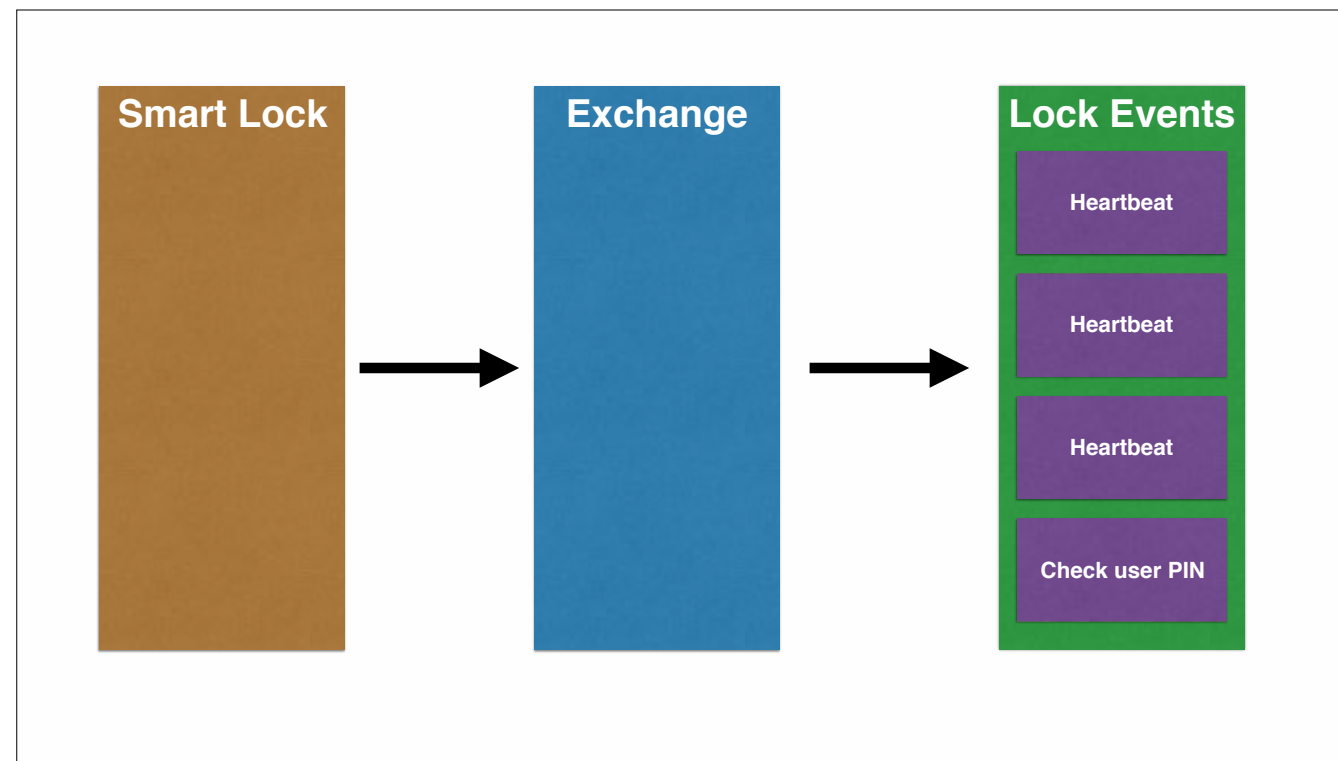
A heartbeat is not that important of a event that we need to stop the world and process it, so we put it into the queue



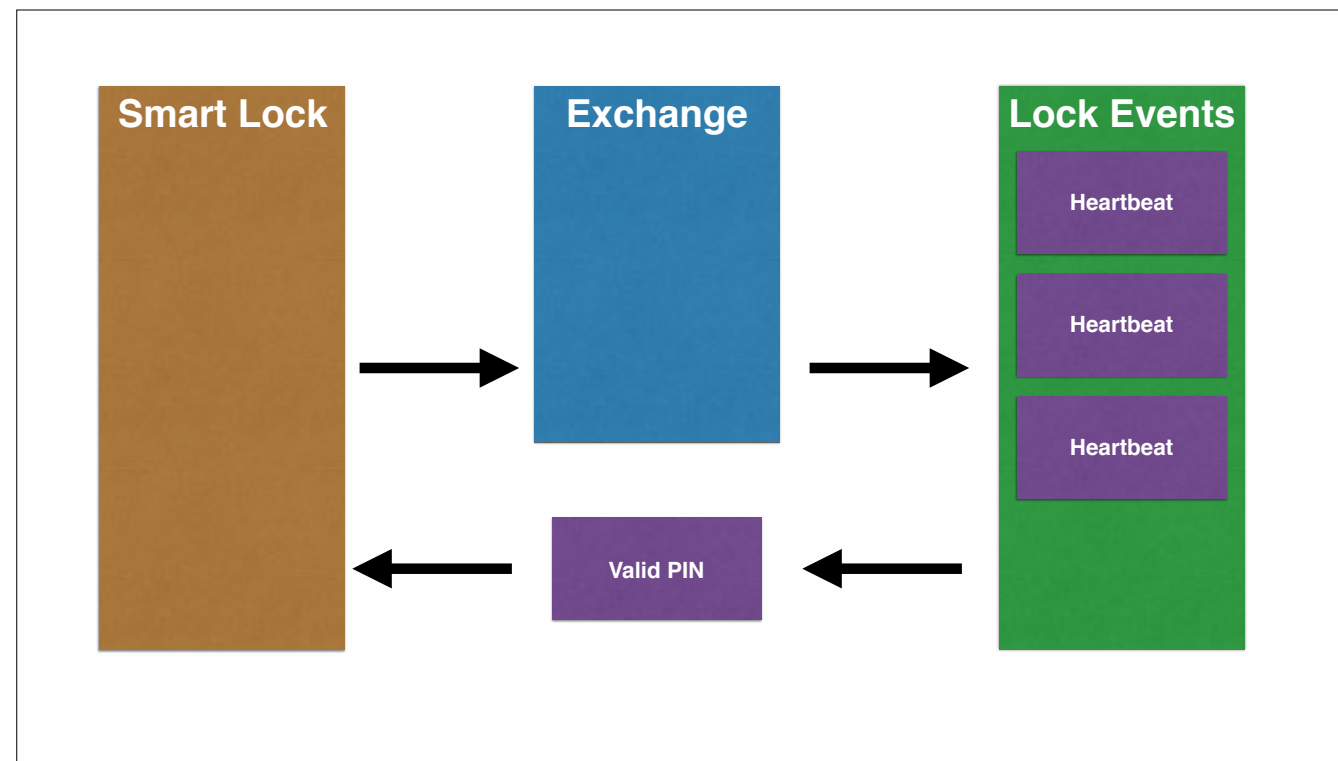
But a PIN check has to be as fast as possible. We don't want to penalise the user for our server load.



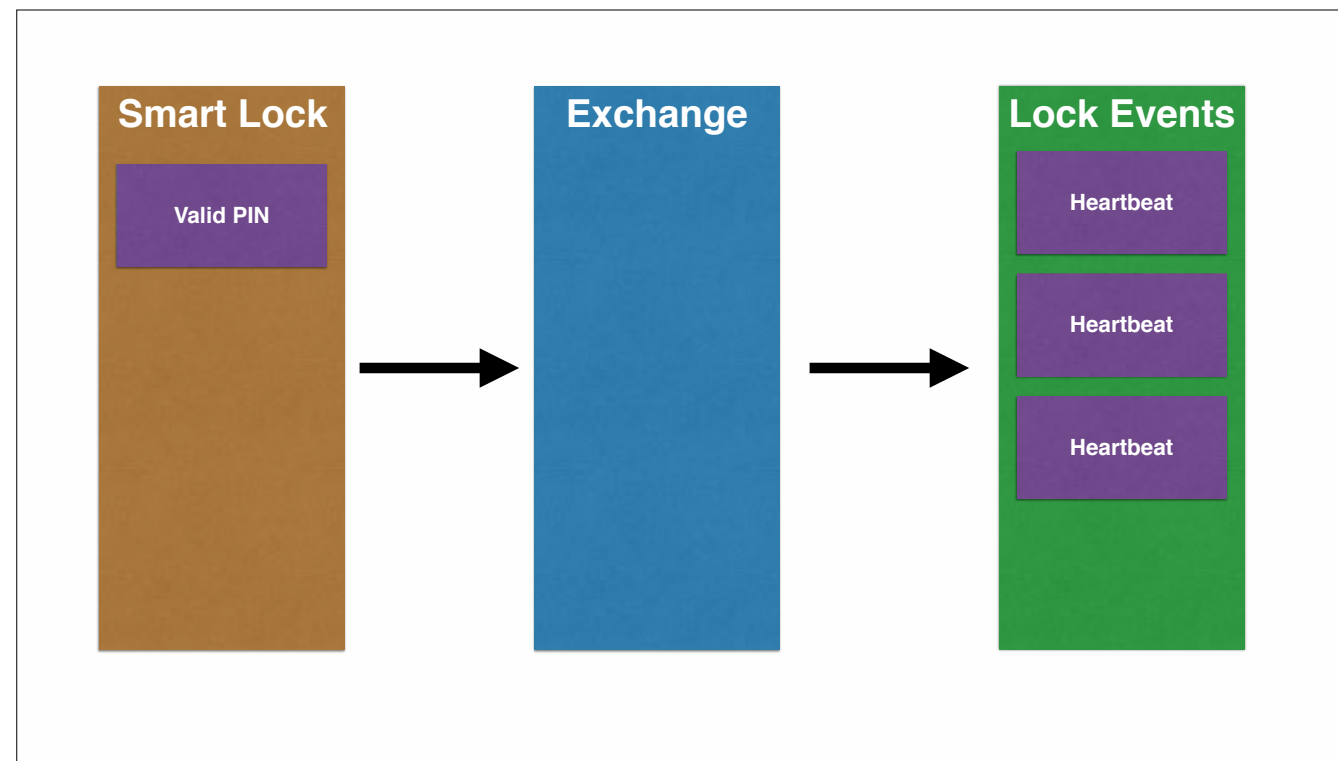
But a PIN check has to be as fast as possible. We don't want to penalise the user for our server load.



But a PIN check has to be as fast as possible. We don't want to penalise the user for our server load.



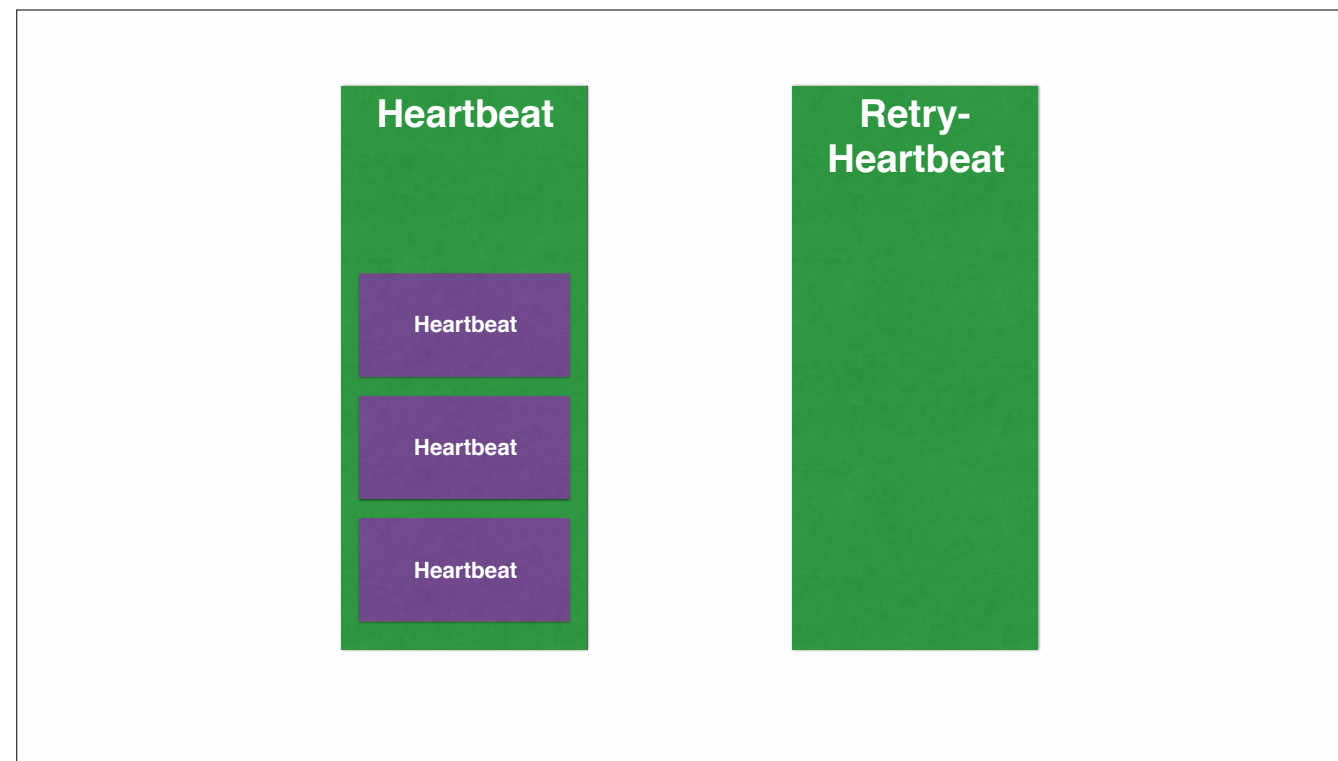
But a PIN check has to be as fast as possible. We don't want to penalise the user for our server load.



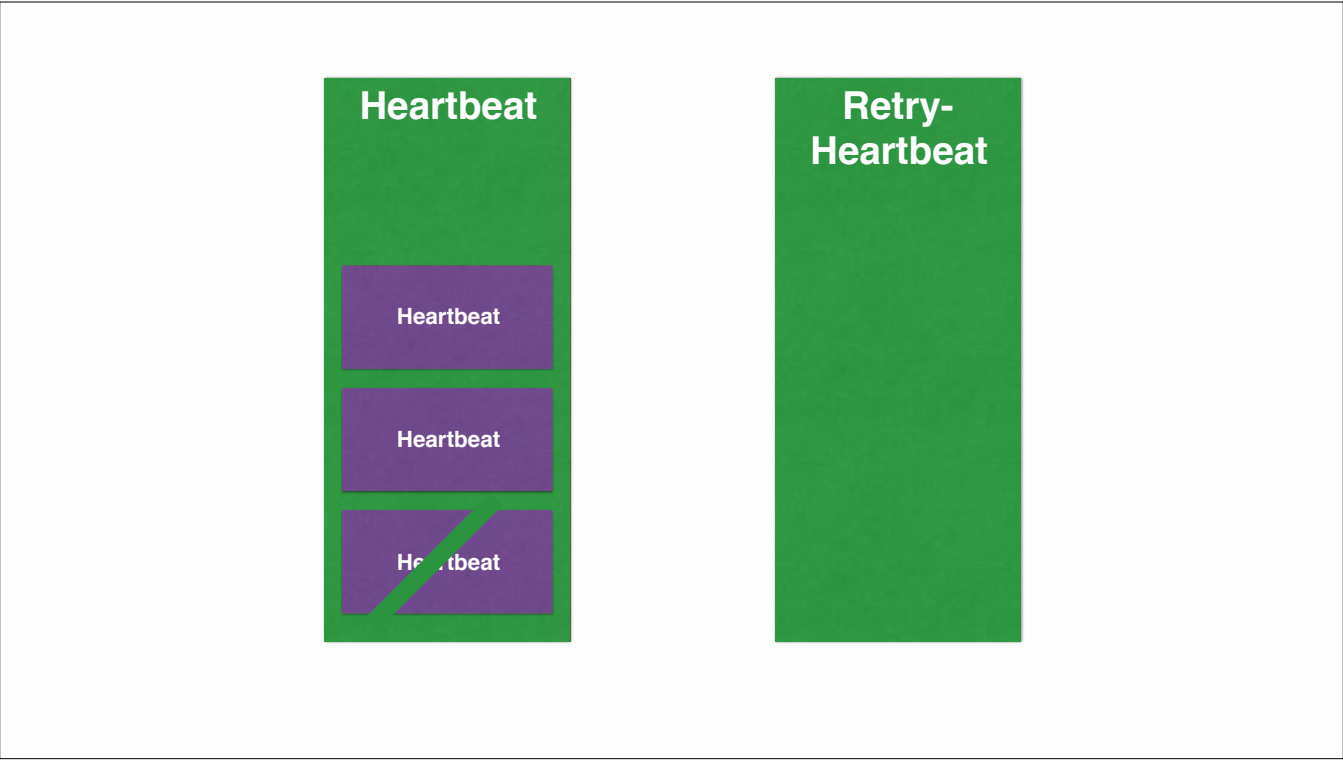
But a PIN check has to be as fast as possible. We don't want to penalise the user for our server load.

Dead lettering

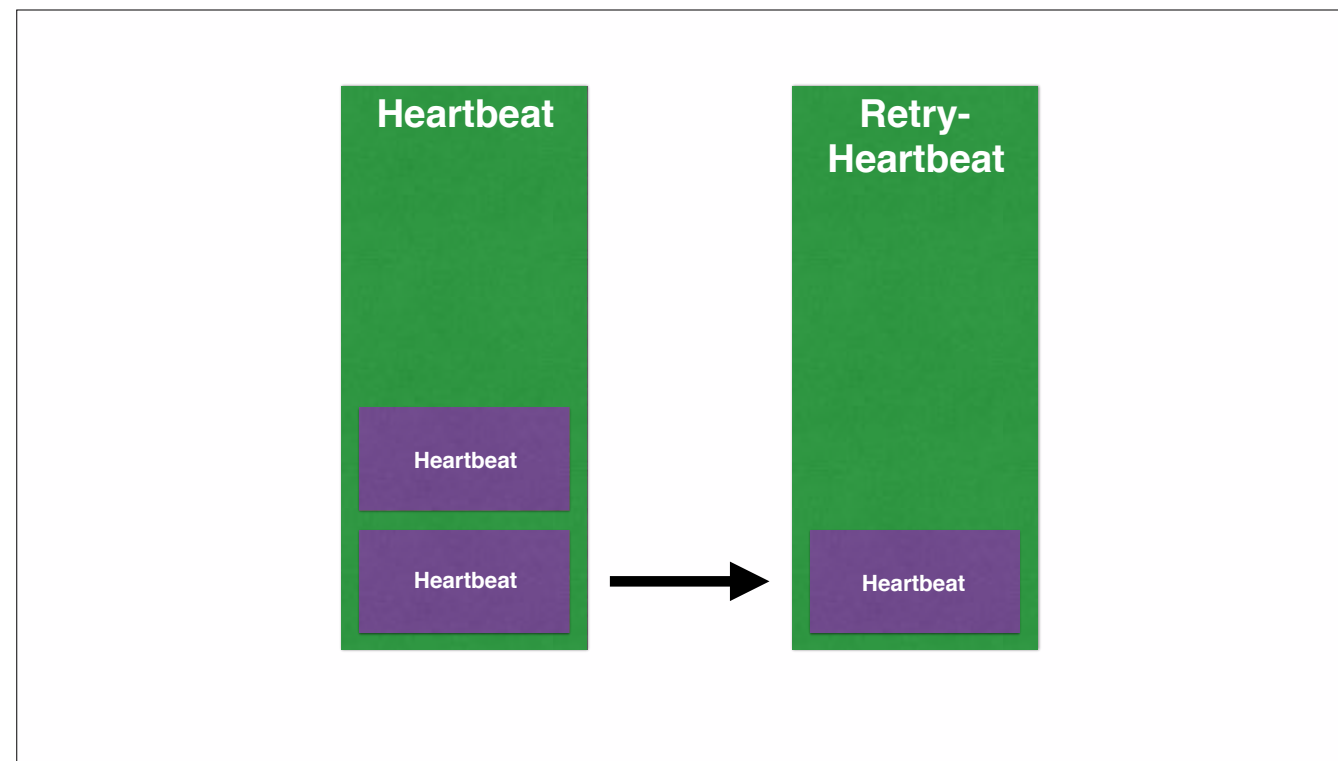
Dead lettering is a fault-tolerance feature for your application. If a message fails for any reason it will get sent to it's corresponding dead-letter queue/exchange.



Now we have two queues. One is our main queue. And the other is our retry queue.



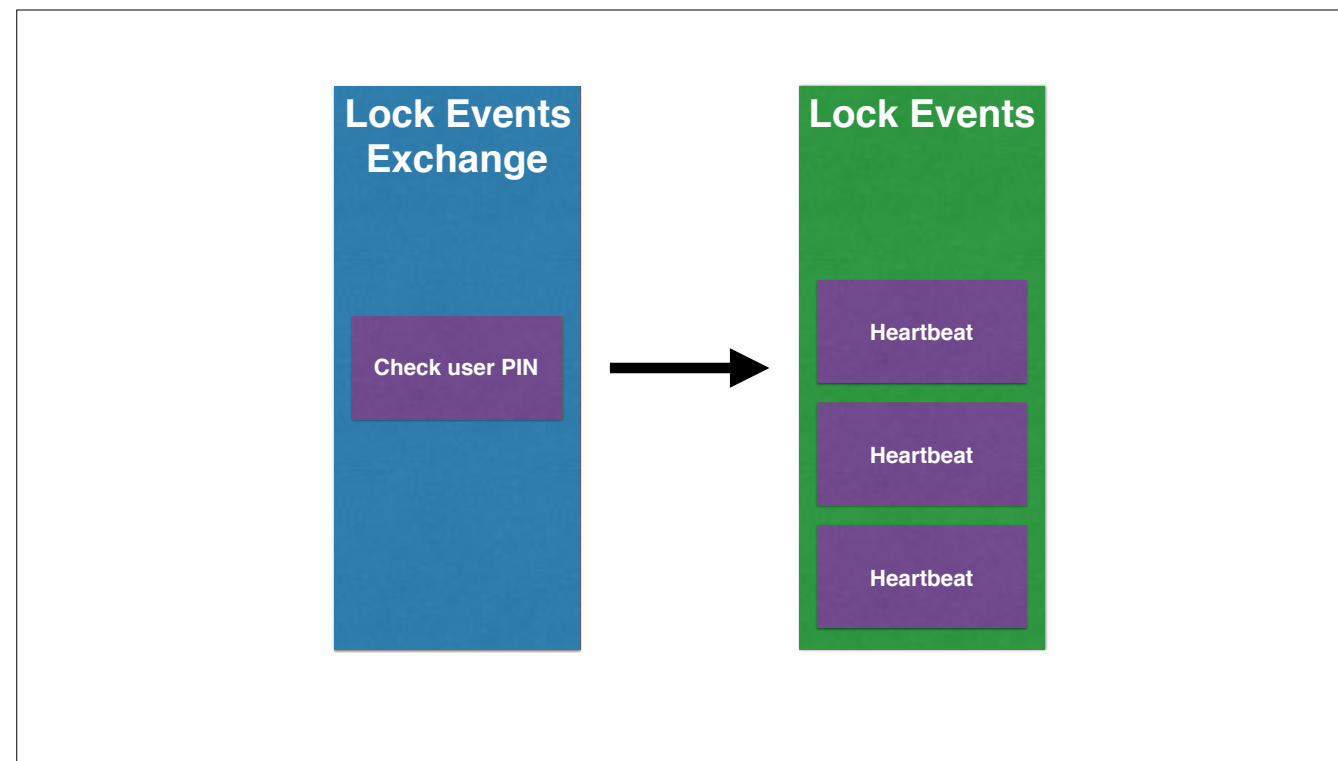
If a job in the main queue fails



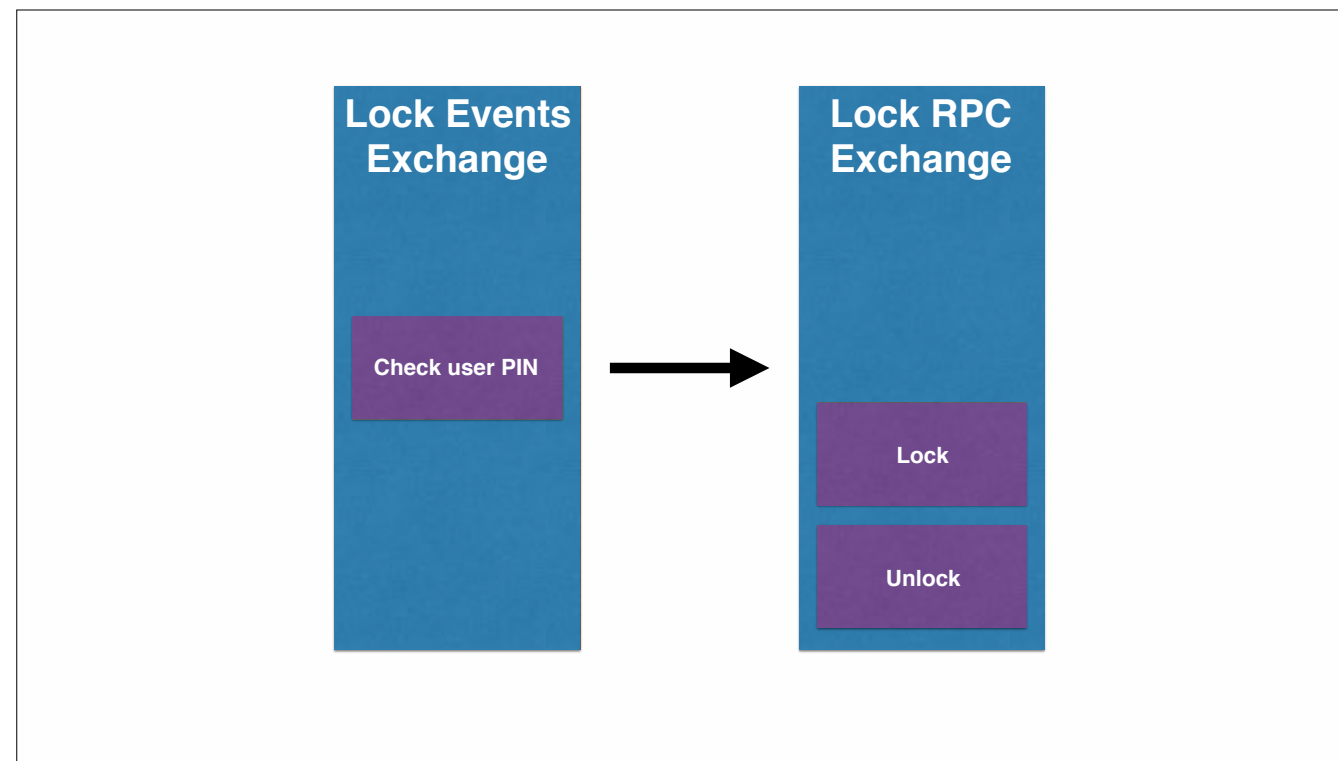
It's transferred to the retry queue. Now we can apply custom logic to handle this failure.

Alternate exchanges

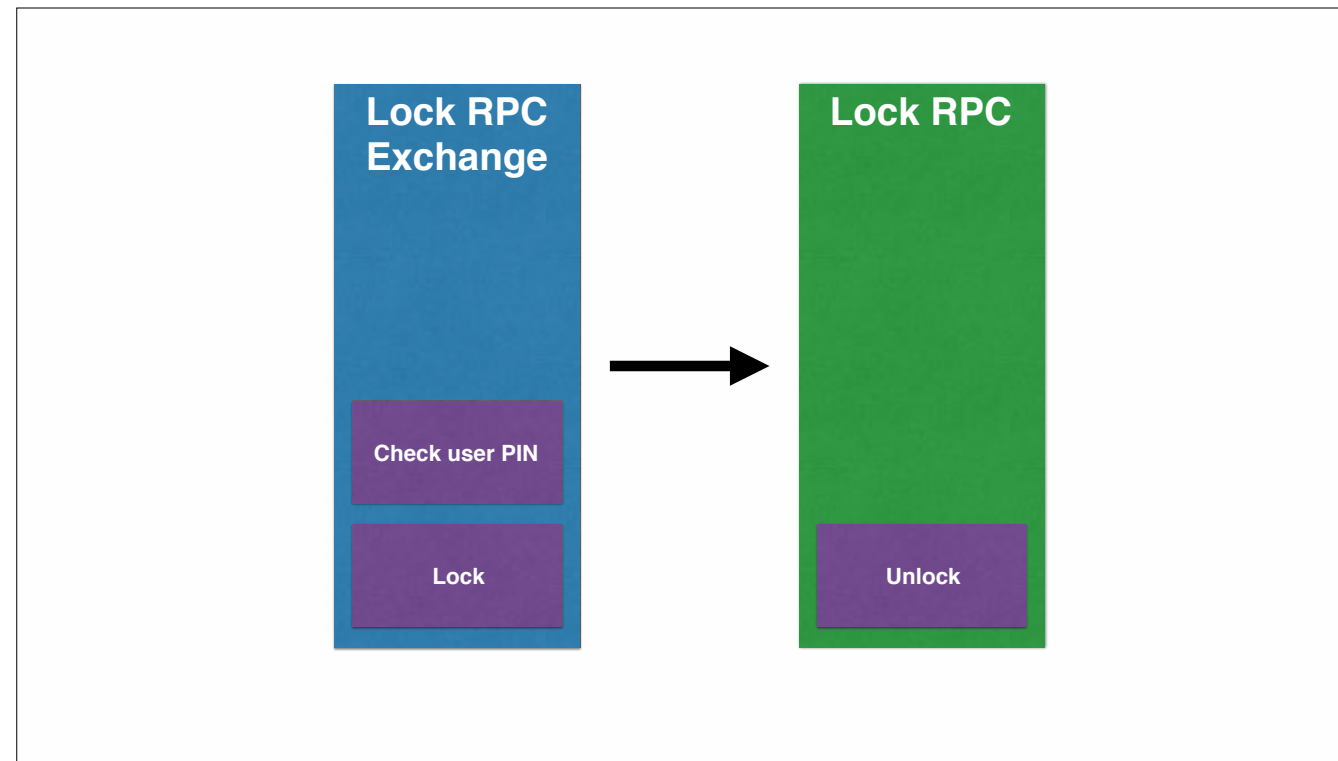
Allows you to specify an alternative exchange to send to if the primary exchange rejects the message



Let's say that the lock events exchange its a topic exchange. If no topic corresponds to the message we sent.



Then the message is routed to the alternate exchange.



Then the alternate exchange handles the message as usual. This feature is quite useful for deprecating services, and rerouting messages of deprecated or legacy services.

Priority consumers

Basically you can say that one consumer can take more jobs than any other. This is useful if your app is running on a diverse set of machines. You can specify that higher capacity machines should get more jobs.

Priority queues

Basically queues where messages with higher priority get processed sooner.



TTL

And TTLs which specify an expiry time for the message.

04+

PLUGINS

Management

Overview

Queue

Queue: messages (queue last message) 1/1



Ready 0
Unacked 0
Redelivered 0

Message stats (queue) 1/1



Unacked 0
Publisher 0
Delivery 0

Consumer 0
Delivery 0
Redelivered 0

Get 0
Get (from ack) 0
Acknowledge 0

Queue stats 0
Ack 0
Queue 0

Global stats 0

Connections 12 Channels 12 Exchanges 12 Queues 0 Consumers 0

Node

Node: rabbit@192.168.1.100 (Node status: OK)

File descriptors	Socket descriptors	Open processes	Memory	File space	Queue space	Queue size	Queue	Queue status
1024	1024	1024	1024	1024	1024	1024	1024	OK

Queue status: OK

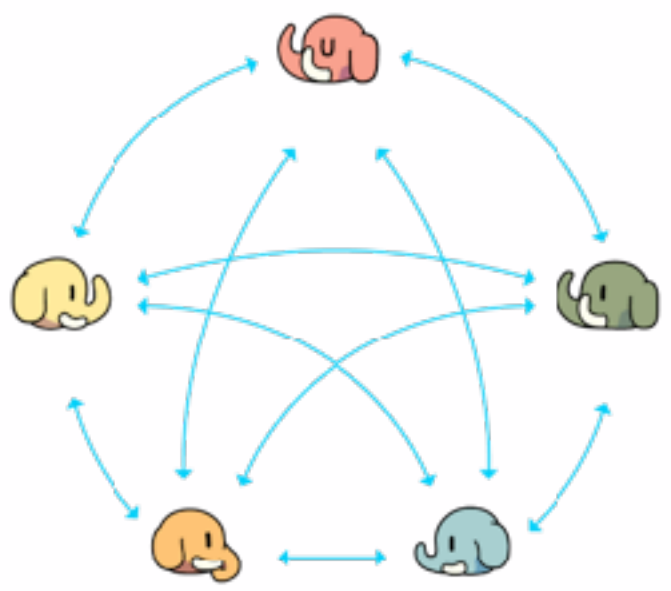
Queue

Config file: /etc/rabbitmq/rabbitmq.conf
 Environment file: /etc/rabbitmq/rabbitmq-env.conf
 Log file: /var/log/rabbitmq/rabbitmq.log
 Data file: /var/lib/rabbitmq/mnesia/rabbitmq

File: /etc/rabbitmq/rabbitmq.conf

MQTT, STOMP and WebSockets

Federation



05

CONCLUSION

Debugging

Load balancing / Scaling

Memory consumption

Should I use it on my next
project?

Monolith?
Probably not...

Except if you need assurances that your jobs will run.

Services?
YES!

QUESTIONS

 github.com/stankek

 [@monorkin](https://twitter.com/monorkin)

 hi@stanko.io

Thank you for your patience. If anybody has questions I'll accept them now.
If you are shy, you can also contact me via Twitter or Email.