# Performance test user behavior model

When designing the performance test, we tried putting together a test case that uses most business functions of the application in a distribution that tries to resemble real user behavior. The decision tree below describes our current test case.

A diagram of a flowchart

Description automatically generated

To give a short summary:

* 5% of users tries to create payment without an access token => The request fails
* 95% of users requests a token and continues the flow
  + 10% of the remaining users sends a request with some validation errror => The request fails
  + 90% of the remaining users sends a payment request with valid data
    - 95% of the time users will attempt to complete payment for these requests
      * 50% of the time it will successful (GIRO ACCEPT)
      * In the other 50% it won’t be successful (GIRO RJCT)
    - 5% of the time users will cancel the payment request before any payment is attempted
    - Regardless of the previous decison 10% of the time users will query the status of the payment

The distribution percentages can be modified anytime, this is just a first draft. Also, extra steps can be added to this process, and we encourage anyone who better understands the real-world use cases to come forward with improvement ideas to this process.

# Testing setup

We wanted to test the performance of the application on the SIT (10.10.70.4) server, because this is also where the GIRO pilot test will take place. Currently we’re using the latest version (5.6.2) of JMeter as a performance testing tool. To avoid network bottlenecks during the test we are running JMeter from the DEV machine (10.10.70.2) that is on the same network, and because it has enough resources to simulate a large number of concurrent users (up to 500) for the test.

# Endpoints

Each step in the decision tree corresponds to an endpoint in the application, these are listed here with the data used during the testing also specified. All tests were performed with the same technical aggregator and sub aggregator, in the future we will try to randomize these as well.

* **Get token: GET** <https://auth-pilot.innopay.hu:8443/realms/innopay/protocol/openid-connect/token>
  + Request a JWT Bearer access token from the Keycloak server
  + Client sends a certificate in order the get a token from the server
* **Create payment: POST** <https://paymentapi-pilot.innopay.hu:8001/api/payment/v1/subaggregator/>sav1
  + Created a payment request with random, formally correct data
  + Dates will be correct dates in the JSON, all fields will be filled out with data resembling real user info (IBAN, ShopId etc.)
* **Cancel payment: PUT** [https://paymentapi-pilot.innopay.hu:8001/api/payment/v1/subaggregator/sav1/payment/{{uti}}/cancel](https://paymentapi-pilot.innopay.hu:8001/api/payment/v1/subaggregator/sav1/payment/%7b%7buti%7d%7d/cancel)
  + Cancel a previously created payment
  + {{uti}} will be filled out with value from the create payment response
* **GIRO ACCP/RJCT: POST** <https://rtp-test1-pilot.innopay.giroinstant.giro.hu:8002/service/recon>
  + Send Pain002 message in XML format
  + In the future we will also test sending Pain002 messages in ASN1 format with the correct certificates
  + Data inside the message will match one of the previously created payments
* **Get payment status: GET** [https://paymentapi-pilot.innopay.hu:8001/api/payment/v1/subaggregator/sav1/payment/{{uti}}/status](https://paymentapi-pilot.innopay.hu:8001/api/payment/v1/subaggregator/sav1/payment/%7b%7buti%7d%7d/status)
  + {{uti}} will be filled out with value from the create payment response

# Collected metrics

After each test we collect the following main metrics to evaluate the results. These metrics will also be available for each endpoint separately.

* **Average Throughput (Hit/s)**: The number of requests the application can serve per second
* **Error (%)**: What percentage of the requests had an error response
* **Average response time (ms):** The average time it took for the application to respond to a request
* **90% response time (ms):** Means 90% of the response times were faster than this. It’s a better performance metric that the average because it eliminates unusual spikes from the result.
* **Min response time (ms)**
* **Max response time (ms)**

A graph with different colored lines

Description automatically generated**A screenshot of a computer

Description automatically generated**To display these metrics in an easily consumable format, we upload the JMeter test results to the Blazemeter site. The results will be available in the following format.

# Load test

A load test is a type of performance test that checks how applications function under the projected number of concurrent users performing transactions over a certain period of time. This is our most basic test case, it simulates a level of load that we should be able to handle without issues during normal operation.

A graph with a red line

Description automatically generated

Max user count: 250  
Ramp up time: 10 min

Steady state: 30 min

Duration: 50 min

# Load test run results - 2023-09-08

## Detailed report can be found here

https://a.blazemeter.com/app/?public-token=VMS0izD9tKvWFJzBY1qqnuOUomicu0kp8R49EZBLAgvaPtTYs2#/accounts/1730298/workspaces/1793262/projects/2081447/masters/69518512/summary

Results Summary  
**Average Throughput: 83.25 Hit/s**

**Error: 0%** (among valid requests)  
**Average response time: 2409 ms**

**90% response time: 9617 ms**

## A graph of a graph Description automatically generated with medium confidenceResource usage

## Conclusion

* Payment request endpoints (create, cancel, getStatus) are stable and could handle more load than currently tested
* We should improve the scalability of Pain002 message processing, because currently that’s the bottleneck in the system
* There are requests that are significantly faster than others, should scale services accordingly
* 250 concurrent users is a bit too much for the system right now, with lower user count (around 100-150) response times are significantly lower

# Load test run results - 2023-09-09

## Detailed report can be found here

## <https://a.blazemeter.com/app/?public-token=EYgVcC4BTtqZixKJVP1KxnoDb5zo5gb3Ojfwzbh21pwhTuDDsz#/accounts/1730298/workspaces/1793262/projects/2081447/masters/69525691/summary>

## Modifications since the previous version

* Removed searching in Redis cache by prefix/suffix (SCAN/KEYS redis command), because as the cache became larger this operation became significantly slower
* The complete pain message was previously stored in Redis, but it was not really needed to accomplish current functionality, so it was removed to reduce cahce size

Results Summary  
**Average Throughput: 144.15 Hit/s**

**Error: 0.000175%** (among valid requests)  
**Average response time: 1388ms**

**90% response time: 4434ms**

## A graph of a graph Description automatically generated with medium confidenceResource usage

## Conclusion

* Throughput improved significantly since last version
* Database seems to be bottleneck, needs to investigate queries further
* GIRO endpoints are still slower on average,
  + GIRO ACCP/RJCT: 3500 ms
  + Create payment: 1085ms
  + Cancel payment: 1494ms

# Step up load test

The step-up performance test is helpful to identify the performance of an application at varying loads. Step-up test starts with a smaller number of users, keeps the load the same for a certain period and then increases the load to the next level. It is very useful when we don’t have exact requirements about how many users we need to handle, because after performing this test we can report how the system behaves under different levels of load.

**Purpose:**

* Collect performance metrics for various load levels
* Based on the metrics identify how many users can we support while keeping response times acceptable

A red line graph on a white background

Description automatically generated

Max user count: 500  
Number of steps: 10

User increase with each step: 50

Steady state in each step: 5 min

Duration: 51 min 30s

# Spike load test

Spike Test refers to a performance test which simulates a sudden high load on the server for a shorter period of time. This helps to identify the behavior of an application when an unexpected huge load arrives. The outcome of the spike test concludes whether the application can handle a sudden load or not.

**Purpose:**

* Check how much the response time increases during a spike
* Identify how much time the system needs to recover after a spike
* Make sure that sudden spikes don’t cause any unexpected errors

A graph showing a line

Description automatically generated

Max user count: 250  
Number of spikes: 4

Spike size: 100 users

Duration: 35 min 30s

Pause between spikes: 5 min

# Resource usage

We monitor the resource usage of each component in the application during each performance test. For this we use Prometheus and various exporters to collect information. The main components monitored are the following.

* **Aggregator API:** The service that handles creation, cancellation and status request of payments
* **SOAP API:** The service that accepts Pain002 messages
* **IQR DB:** PostgreSQL database
* **Keycloak:** Identity and Access management server
* **Redis:** Distributed cache solution

For each component we collect the following metrics:

* **CPU usage**
* **Memory usage**
* **Received/Sent network traffic**
* **Disk I/O operations**

Sample resource usage metrics

A graph of different colored lines

Description automatically generated

A graph of a graph

Description automatically generated with medium confidence