Final thesis

A scalable back-end system for web games using a RESTful architecture

by

Emil Helg and Kristoffer Silverhav

LIU-IDA/LITH-EX-A--16/045--SE

2016-07-25
Final thesis

A scalable back-end system for web games using a RESTful architecture

by

Emil Helg and Kristoffer Silverhav

LIU-IDA/LITH-EX-A--16/045--SE

2016-07-25

Supervisor: Aseel Berglund
Examiner: Henrik Eriksson
Abstract

The objective of this thesis was to design and implement a scalable and load efficient back-end system for web game services. This is of interest since web applications may overnight gain a significant increase in user base, because of viral sharing. Therefore designing the web application to service an increasing amount of users can make or break the application, in regard to keep the user base. Because of this, testing how well the system performs during heavy load can be used as a foundation when making a decision of when and where to scale up the application. The system was to be generically accessible through an Application Programming Interface (API) by the different game services. This was done using a RESTful architecture where emphasise was put on building the system scalable and load efficient. This thesis focuses on designing and implementing such a system, and how load testing can be used to evaluate this systems performance for an increasing amount of simultaneous clients using the web application. The results from load testing the implemented system was above the expectations, considering the hardware used when running the tests and hosting the system. The conclusion of this thesis is that by following REST when designing a web service, scalability becomes a natural part of how one would design the system.
Acknowledgments

We would like to thank Aseel and Erik Berglund at Linköping University for the opportunity of doing this thesis and their continuous help and support. We both would like to thank our friends and families for believing in us and giving us the strength to get this far.
# Contents

Abstract iii  
Acknowledgments iv  
Contents v  
List of Figures vii  
List of Tables viii  
Acronyms 2  

## 1 Introduction 4  
1.1 Motivation ................................................. 4  
1.2 Aim .................................................. 5  
1.3 Research questions ...................................... 5  
1.4 Delimitations ............................................ 5  

## 2 Background 6  
2.1 Bug game .................................................. 6  
2.2 Phaseformer ............................................... 7  

## 3 Theory 8  
3.1 Web technologies ........................................ 8  
3.1.1 Hypertext transfer protocol .......................... 8  
3.1.2 Representational state transfer ....................... 8  
3.1.3 Platform as a Service ................................ 10  
3.2 Security aspects in web applications .................... 10  
3.2.1 Confidentiality, integrity and availability ......... 10  
3.2.2 Transport layer security ............................. 10  
3.2.3 Cross-site request forgery ........................... 11  
3.2.4 Payment vulnerability ................................ 11  
3.3 Technologies .............................................. 11  
3.3.1 Python ................................................ 12  
3.3.2 Flask .................................................. 12  
3.3.3 OpenShift ............................................. 12  
3.3.4 SQLAlchemy ........................................... 12  
3.4 Scalability ............................................... 13  
3.4.1 Definition ............................................ 13  
3.4.2 Vertical scaling ..................................... 13  
3.4.3 Horizontal scaling ................................... 13  
3.4.4 Optimization of code ................................. 14  
3.5 Testing .................................................. 15
### Method

4.1 Agile development

4.1.1 Iterative development

4.1.2 Requirement elicitation

4.2 System design

4.2.1 Designing the database

4.2.2 Designing the application programming interface

4.2.3 Security

4.3 Implementation

4.3.1 Users and user relationships

4.3.2 Store and payment service

4.4 Load testing

### Results

5.1 Requirement elicitation

5.2 System design

5.2.1 Database

5.2.2 Application programming interface

5.3 Implementation

5.4 Load testing

### Discussion

6.1 Results

6.1.1 System design

6.1.2 Load testing

6.2 Method

6.2.1 Agile Development

6.2.2 System design

6.2.3 Implementation

6.2.4 Load testing

6.3 The work in a wider context

### Conclusion

7.1 Future Work

### A Requirements

### B Tsung Test Suite

### C Test Cases

### Bibliography
## List of Figures

2.1 A screenshot of the bug game while playing ........................................ 6  
2.2 A screenshot during a playthrough of the Phaseformer game .................. 7  
3.1 Difference between horizontal and vertical scaling ............................. 14  
5.1 The three system layers ................................................................. 26  
5.2 EER diagram over the database ....................................................... 27  
5.3 A code snippet creating a database model ....................................... 31  
5.4 A code snippet showing how routing is done .................................... 32  
5.5 A code snippet showing how information about a user is sent ................ 32  
5.6 A code snippet showing how form data is validated ............................ 32  
5.7 Test output from testing the system on OpenShift using the test case described in Appendix C.1 ................................................................. 33  
5.8 Test data from running two different test cases on the localhost with three phases each. With the pictures to the left representing data from the test case described in Appendix C.2 with each phase being 400 seconds long. The pictures to the right representing the test case described in Appendix C.3 with each phase being 600 seconds long. ................................................................. 34  
5.9 Test data from running two different test cases on the localhost with six phases. The third phase is twice as long as the rest of the phases to accommodate a cooldown time. With the pictures to the left representing data from the test case described in Appendix C.4 with each phase being 120 seconds long. The pictures to the right representing the test case described in Appendix C.5 with each test case being 240 seconds long. ................................................................. 35
List of Tables

5.1 A table showing the routes and HTTP method applicable for the user resource . . 28
5.2 A table showing the routes and HTTP method applicable for misc. resources . . 30
5.3 A table showing the routes and HTTP method applicable for the clan resource . . 30
5.4 A table showing the routes and HTTP method applicable for the store resource . . 31
Acronyms

API  Application Programming Interface
BSD  Berkley Software Distribution
CIA  confidentiality, integrity and availability
CPU  central processing unit
CSRF Cross-Site Request Forgery
DBMS database management system
DDL  Data Definition Language
EER  Enhanced Entity-Relationship
FTP  File Transfer Protocol
GUI  Graphical User Interface
HTML HyperText Markup Language
HTTP HyperText Transfer Protocol
HTTPS HyperText Transfer Protocol Secure
JDBC Java Database Connectivity
JSON JavaScript Object Notation
ORM  object-relational mapper
PaaS Platform-as-a-Service
QoS  Quality of Service
RAM  random access memory
REST Representational State Transfer
ROA  resource-oriented architecture
SOAP Simple Object Access Protocol
SQL  Structured Query Language
SSL  Secure Socket Layer
TLS  Transport Layer Security
URI  Uniform Resource Identifier
WSGI  Web Server Gateway Interface
XML  Extensible Markup Language
Chapter 1
Introduction

Chapter one will focus on the motivation and aim behind this thesis. Defining what is of interest and the limitations placed on the work.

1.1 Motivation

With the introduction of the smartphone and tablet, gaming has gone from being played on dedicated systems, such as video game consoles or dedicated computers, to being able to be played at almost anytime and anywhere by anyone. In the US alone there existed an estimated 170 million gamers during 2013, of these 91 million played using a smartphone [31].

During the same time, social networking was growing at a fast pace. This growth lead to people sharing more content between each other, which soon encompassed gaming related content. Allowing gaming companies such as Zynga to create games based on sharing their experiences with their friends through these social networks.

This shift in where we play games, how we play them and the sharing of content has lead to the business model of releasing free games and then using in game purchases as a source for revenue. Making use of this, developers can now develop games without the need of a publisher to print hard copies of the game, but instead being able to publish the games on the different devices app store. The sharing of content in the games, allowed for some games to gain a quickly growing user base. With users sharing their accomplishments to friends and family, motivating them to share their experiences within the games.

The developer community have also been busy developing new frameworks and tools to allow easier and faster development of games for a multitude of platforms. The introduction of version 5 of the HyperText Markup Language (HTML) has also speed up this trend, by supplying an increasingly more adaptable and powerful web platform that all devices are able to make use of.

A problem with this fast expansion is the lack of academic work being done on the subject of back-end functionality for games, especially regarding the technical aspects. Another problem lies in the scaling of the back-end when the clientele increases. That is to be able to handle an increasing amount of users, while the user experience stays within an acceptable range regarding things such as, the response time. One of the ways of designing the system to accommodate for the scaling, is to use a RESTful architecture for the back-end system. This thesis will try to add to the subject, by showcasing all the steps from designing the architecture to implementation of a system. This systems functionality will handle payment logic and state description for game accounts, among other things. This will be done by supplying a generic API, that web games can make use of.
1.2 Aim

The purpose of this thesis is to design and implement a scalable and load efficient back-end system. This system will supply payment functionality for web based game services. The efficiency will be from the system being able to handle increasing amount of users with little degradation in service. The purpose of this system is also to be generic, so as to provide its functionality irregardless of the platform the game is running on, and how the game is designed. The main subject of this thesis will be the design and implementation of a scalable and load efficient back-end service for game clients. This goal will be evaluated by load testing the server that the system runs on, to see how the performance of the system degrades.

1.3 Research questions

These are the following research questions that are of interest, and will be answered by the end of this thesis.

1. How can a RESTful architecture be implemented for a back-end server to emphasize scalability?
2. Using this approach, how many concurrent users can be serviced within an acceptable response time?

Scalability in this context refers to the ability to handle an increasing amount of load on the system. 15 seconds is considered the longest time a client is willing to wait for a response from the server [35]. Therefore an acceptable response time for this thesis is 15 seconds.

1.4 Delimitations

The system will be developed using Python with the Flask framework, using PostgreSQL as the database management system (DBMS), payments will be handled with Stripe as the service provider. The usage of Python and Flask was placed by the stakeholders, which are the product owners. The usage of PostgreSQL and Stripe, was decided in conjunction with the stakeholders.
Chapter 2
Background

This chapter will briefly describe the two web games being developed at Linköping University that the system will be designed to receive requests from, and how the work done in this thesis will be used by them.

2.1 Bug game

The Bug game is a motion controlled game run in the web-browser, where the user is supposed to squash insects to hinder them from reaching a tree. Since if a bug reaches the tree, the level will end and the user will not be able gain more score on that try, and the user has to restart the level and make a new try for a higher score. The user must also take care to not squash water drops as this damages the player and after enough damage the game ends. The game is played with the usage of a camera to capture the users motions as input. The game will make use of a payment system, wherein a user is able to pay money to receive virtual items. Both the user progress as well as the payment will be done using the back-end system developed in this thesis.

Figure 2.1: A screenshot of the bug game while playing
2.2 Phaseformer

Phaseformer is a 2D platform game developed in phaser.io. The game make use of a camera for input, eschewing the more common keyboard or gamepad. The character can be controlled by the user motioning above certain defined areas of the canvas to move the game character. The goal of the game is for the player to make his way through the level, while collecting coins as he goes. The progress of a user, will be stored in the back-end system.

Figure 2.2: A screenshot during a playthrough of the Phaseformer game
Chapter 3
Theory

This chapter will describe the basis upon which the work of this thesis will build on, describing the technologies, concepts and protocols used during this thesis. It will also contain some background on the economical aspects in games.

3.1 Web technologies

This section will describe the terminology and technology used regarding the protocols and architecture related to the web.

3.1.1 Hypertext transfer protocol

HyperText Transfer Protocol (HTTP) is used as a method for encoding and transporting information between a client and a server sending information between them over a network. HTTP works by using a request/response architecture, where the client send a formatted request message to the server, and the server responding with a formatted response message. The response message from the server contains among other a status code in response to the request [27]. This status code represents the server’s attempt to understand and process the request. The status code is described with a three-digit integer, with the first digit defining the class of the response, and the last two numbers specifying the type. With the introduction of HTTP/1.1, requests can be made using the same connection, no new connection is needed for each request/response exchange [27]. A HTTP packet request contains among other a method header. This header defines the operation a receiver should do upon the referenced resource. The four most used method headers are: GET, PUT, POST and DELETE.

3.1.2 Representational state transfer

Representational State Transfer (REST) in accordance with a resource-oriented architecture (ROA) defines an client-server architectural style, with given constraints on the application [44]. The given constraints are adherence to a client/server architecture, uniform interface, statelessness, cacheability, and layered systems [21].

Uniform interface

One of the most fundamental constraints and important designs in a REST application is the uniform interface. The uniform interface describes how the interfaces for the resources supplied to the client by the server, are uniform in that the Uniform Resource Identifier (URI) describes the path to the resource. Then the method header in the communication protocol should describe the action to be taken [44].

An example of this would be using either GET, PUT, POST or DELETE on the URI of example.com/forum/thread-1. This would, depending on the HTTP method header act differently, as shown below.
3.1. Web technologies

**GET** The server will try to retrieve data from this location, this will often be a HTML document. In the given URI example, the data retrieved would be the content of the forum thread.

**PUT** The server will either create a new thread in the forum named thread-1 or, if a thread with that name already exists, modify it.

**POST** The server will append a comment to this thread.

**DELETE** The server is expected to delete the thread and all the comments in it.

All of the actions performed on the server should be made depending only on the method header, the interface to the resources in the form of the URI stays the same.

**Statelessness**

Having an stateless application in the context of REST, means that all the request received on the server happens in isolation. With the client supplying all the information needed in the request. The server never tracks previous actions from the client and expects that the client supplies state specific information if needed. This means that all resources on the server are addressable through an URI. With state specific information being sent in the request, the order of requests is of no importance to the server [44]. This statelessness leads to all possible states on the server being reachable and addressable for the client, as long as the client are able to supply the correct information for that specific state.

**Cacheability and layered systems**

A cacheable application, is an application in which previous results from requests can be saved in caches so that the response can be used by later requests [21]. A response to a request can be cached at another location in a distributed network, such as the Internet. Caching is used both as a way of speeding up the response time for requests, since a cached copy of the resource may be found earlier. But it can also be used as a way of handling more clients to a server, by spreading common request responses at different locations, reducing the load on the main server application.

A major strength of REST is its simplicity. An application that follows the constraints given by REST can be created without much effort, an understanding of REST and a library for communication via HTTP can get a programmer far. As pointed out by Richardson et al [44], the complexity of services not following the REST constraints is often made unnecessarily complex. The service can become hard to debug, maintain, and may not work if the clients accessing the service are not correctly setup.

A major concern regarding REST, lies in the difficulty of representing bi-directional communication. Or as pointed out during a discussion whether REST is possible to achieve using websockets [57] a commentator pointed out that:

“If you consider REST in the Fielding sense, with a web of addressable objects (or resources), then that doesn’t really work in a duplex comm format. You don’t expect the resources to initiate the conversation. WebSockets will transform the web (if they take off), but not as a protocol for REST-style communications.”

In the context of this thesis this is not a problem, because bi-directional communication is not necessary.
3.2. Security aspects in web applications

3.2.1 Confidentiality, integrity and availability

Information security can be based on three key concepts, confidentiality, integrity and availability [6].

Confidentiality represents the need to keep information hidden or as a secret. This can be implemented in a variety of ways depending on the needs. The knowledge of the existence of the information can be hidden to a few users/clients. Access to the information can only be given to a certain few, blocking the access for the rest. Or the information can be stored so all can access it, but by using cryptographic methods to make the information unusable to all but the few with the means of deciphering the information.

Integrity is a measure of the trust on the information. Protecting the integrity of the information, is the aim of preventing improper or unauthorized change of the information. Integrity protection can either be done proactively or reactively. Prevention mechanisms tries to prevent improper or unauthorized changes to occur. Detection mechanisms, as the name implies tries to detect if the informations integrity has been breached.

Availability refers to the aspect of being able to access the information. The loss of Availability is because of deliberate attempts from a third party to hinder access to the information.

3.2.2 Transport layer security

Transport Layer Security (TLS), which is frequently and interchangeably referred to as Secure Socket Layer (SSL), is a cryptographic protocol aiming to provide secure communication and is a successor to the SSL protocol [14]. TLS tries to provide this secure communication by ensuring the confidentiality and integrity of the information being sent.

TLS works by using asymmetric encryption to initiate the session. By using asymmetric encryption to allow secure communication and certificates for authentication, the client and server are able to decide on an symmetric encryption algorithm and hash function. After this they exchange random values to create the shared key to be used during the session. When this is done the connection uses the predetermined key and encryption algorithm to ensure the confidentiality of the information being sent. While using the hash function as a means to ensure integrity of the information being sent.

The main drawback of using TLS, lies in the performance cost, increasing the load placed on the server for each client [9]. Christian et al [9] shows that the largest performance cost
is represented in the asymmetric encryption/decryption operations. While this weak point in TLS will have an noticeable impact on the back-end system, there exists not that many choices. TLS is de facto standard for secure communications over the web, and as Christian et al points out; dedicated RSA accelerators or an increase in central processing units (CPUs) or CPU speed are effective to reduce the performance impact.

3.2.3 Cross-site request forgery

Cross-Site Request Forgery (CSRF) was one of the top 10 most critical web application security risks in 2013 [58]. The exploit is about a malicious site instructing a users browser to send an unwanted request to an honest site where the user is authenticated, this makes it look like the user is doing the request to that site [4]. Since the malicious site has no way of seeing the response of the forged request that were sent by the authenticated user, the attack focuses on state-changing requests instead of data theft requests.

It is important for web application to mitigate CSRF exploits to ensure the session integrity for the users. The most commonly used mitigations against CSRF attacks are [41]:

1. To include a secret token with each state-changing request to be validated on the server to see if the token is correctly bound to that users session, if the token fails the validation the request is declined.
2. Validating the HTTP referer header to see which URI initiated the request and declining requests made from untrusted sources.
3. Checking the origin HTTP header which will unlike the HTTP referer header be present in requests that originates from an HyperText Transfer Protocol Secure (HTTPS) URI.
4. Using a challenge-response like CAPTCHA, re-enter password or using one-time tokens for each state-changing request.

3.2.4 Payment vulnerability

A major problem with payment systems via the web, is the vulnerabilities that are often exposed. These vulnerabilities requires not only a secure mindset when implementing the system. It also requires the need for secure technologies, experience, and an understanding of how malicious attempts can be made. The reliance of third-party services that supplies the payment logic to a system can also contain serious logic flaws that can be exploited [56]. Understanding of how some of these flaws may look like, and how they can be exploited should reduce the likelihood of these occurring in systems.

Wang et al [56], describes some real world cases of logic flaws, and how these were found. Using the open-source software NopCommerce, integrated with PayPal Standard, they showed how the payment logic did not check whether the gross value of the purchase is the correct amount when paying the order [56]. A malicious user was therefore able to change the amount taken during payment from his account, paying less or more than the actual cost. They also showed that using the commercial software Interspire, integrated with Google Checkout, there existed a flaw where a user could go to checkout and pay for the item, but an order was not yet generated. The user is then able to keep adding items to the cart, which will be added to the order later generated [56]. Using this method a malicious user could add a cheap item to his cart, checkout that cart, then keep adding items to the cart. So that when paying for the cart, the user would then only pay what the first item cost, irregardless of the new total cost of the newly added items.

3.3 Technologies

The technologies used in this study will be described in this section.
3.3. Technologies

3.3.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Python has a very simple and clean looking syntax. The language uses white-space or tabular indentation which makes the code very clean and easy to read. Python was developed in 1990 by Guido van Rossum.

Python is a platform independent programming language. Running a Python program compiles the code into platform independent byte code, which runs on a virtual machine. The kernel for Python is small, but can easily be extended by importing extension modules or writing modules of your own [47]. The Python standard library offers a variety of standard extensions, among them high-level data types such as collections and arrays, web-related utilities, data compression and generic operating system services [23].

3.3.2 Flask

Flask is a micro framework written in Python for web application development, it is called a micro framework because the aim of Flask is to keep the core simple but extensible. Flask is BSD licensed which makes the framework free of charge, even for commercial purposes [45]. The framework is based on Werkzeug which is a Web Server Gateway Interface (WSGI) utility library that also provides routing and debugging. Flask is also based on Jinja2 which is used for template support in Python.

Flask itself is really small and therefore many of the core services for a web application is not in the native packages, features like accessing databases, authenticate users, validating input in web forms, etc. These core services can be accessed through extensions that integrate with the core packages [26]. Flask gives freedom to the developer, it lets the developer choose exactly what extensions to use or even write own extensions. It supports extensions for relational databases, as well as extensions for NoSQL databases, or even a self written extension for a homegrown database engine. The framework was developed from the beginning with intentions of being extended.

3.3.3 OpenShift

OpenShift is a PaaS. The product provides hosting, developing and scaling applications in a cloud environment. OpenShift was released in 2011 by Red Hat, an American multinational software company that provides open-source software products.

OpenShift supports Python along with a lot of other popular language environments like Java, .NET, Ruby, etc. OpenShift also supports many different frameworks, including Flask but also other popular frameworks like Django and Ruby on Rails. The supported databases are MySQL, MongoDB, PostgreSQL, SQLite and Amazon RDS [42].

There are three different versions of OpenShift; OpenShift Online, OpenShift Enterprise and OpenShift Origin [43]. Applications hosted on the OpenShift Online version runs on the public cloud, this version is free but can be upgraded to premium to get support from Red Hat instead of the community which is for the free version. OpenShift Enterprise requires an annual software subscription and lets the application run on own servers or on a private cloud, this version is provided with support from Red Hat. OpenShift Origin is an open-source project and the application can run on own servers, local machine, private or public cloud. Support for this version is provided by the community.

3.3.4 SQLAlchemy

SQLAlchemy is an Python interface for DBMSs such as MySQL, SQLite and PostgreSQL [11]. Queries for the DBMS can be written using either the Structured Query Language (SQL) expression language, or using the object-relational mapper (ORM) included in SQLAlchemy, allowing for persistent storage of data. Using SQLAlchemy also allows for mapping between
different Python objects being stored in the DBMS. Raising the level of abstraction a programmer can work with, by allowing access to functionality such as inheritance between the data objects. Standard SQL relationships such as one-to-one, many-to-one, and many-to-many can be written using the object oriented approach of employing encapsulation between the objects. SQLAlchemy provides a set of well known enterprise-level persistence patterns which makes the code robust and adaptable. It is using a flexible design that makes it simple to write complex queries to the database. SQLAlchemy has a heavyweight API which can lead to a long learning curve [24].

There are many more ORMs that can be used in Python, some of them are SQLObject, Storm, and peewee. SQLObject has a small code base and uses the active record pattern which does not support database sessions. Storm has a lightweight API and does not need imperative base classes to be able to construct tables from the model classes. Storm has no support to automatically generate the database tables from the model classes, manual Data Definition Language (DDL) statements are required to create the tables. Peewee has a lightweight API and are easy to integrate with any web framework, but it does not support automatic database schema migrations [24].

3.4 Scalability

This section will describe what scalability is and what types of solutions there are to make a system more scalable.

3.4.1 Definition

Scalability is the ability of a system to handle an increasing amount of load. When a system is using a client/server architecture, scalability is limited by how much load the server host can handle [20]. With an improved load capacity, the server is able to serve a greater amount of users with little or no downgrade in user experience. As pointed out by Jeremy et al [19] a service which gains quick recognition via media can be visited by tens of thousands of people. If these peoples first experience is that the service is not working, many people will not bother to retry. There are different types of solutions that can be used independently or in combination to increase the load capacity of a system, solutions like: vertical scaling, horizontal scaling, or optimization of code [19].

3.4.2 Vertical scaling

Vertical scaling is when the hardware of the current system is upgraded to be able to work under more load. The system can scale by upgrading the RAM, hard drive, or exchange the current CPU to a more powerful one on the servers, this increases the capacity of each server. Vertical scaling is often limited to the capacity of a single unit in the system. This scaling solution works in any system, irregardless of how simple or complex the design is, if there are better hardware alternatives. When the hardware limit has been reached, there are no more upgrades that can be made to the current hardware, vertical scaling can not be used anymore to upgrade the system [29]. This type of scaling is time-efficient since no changes need to be done to the software, but it is considered as an expensive strategy in regard of increased performance compared to the cost of the upgrade [48].

3.4.3 Horizontal scaling

Horizontal scaling can be done by adding more servers, creating a cluster of the system instead of upgrading the hardware, by doing so the load on the system can be balanced on many servers. Horizontal scaling does not limit the system to the capacity of a single unit.
This type of scaling requires the system to be more complex [48], implementing a load balancer to distribute the load between the servers, and adding more databases often requires replication of data between these. Building a system that can scale horizontally takes time, but makes up for it in regards of expenses for scaling up the system to handle more simultaneous users.

Another approach to reduce server bottlenecks is to make use of caching content [55]. By placing content responses in proxy caches for common requests, content is spread to other locations outside the system. These requests can then be serviced by the proxy, reducing the amount of requests that is sent to the main system. These proxies can then be placed at different geographical locations, both increasing the likelihood of more clients fetching the response through the cache, and reducing the latency for individual clients. This geographical spreading of the proxy caches, also increases the likelihood of content being accessed more than once by clients [2]. A major problem with using caches for storing content from the web is that caches have finite memory, forcing caches to decide which content to keep, and which to delete when memory is full. Another problem with storing content in caches becomes the relevancy of the stored content. Storing content for a longer time increases the probability of the content not being up to date with the content stored in the system, from which the content comes from [2, 16].

![Vertical and Horizontal Scaling Diagram](image)

**Figure 3.1: Difference between horizontal and vertical scaling**

### 3.4.4 Optimization of code

This can be done in many different ways. One way is to reduce the complexity of the internal logic, improving the data structures or algorithms used [50]. Another way is by removing or reducing the impact of existing bottlenecks in the code, one such bottleneck lies often in the database. With an increased amount of users in the system, more queries will be made to the database by the server. To be able to handle more load, the queries can be optimized. Optimizing the queries will reduce the time it takes to read, insert and delete data in the
3.5 Testing

Testing software can be done in a variety of ways, it can be done informally by exploring the system and its boundaries, or by automation using predefined test cases. Testing can also be done for a variety of reasons. To either being done to assure that a requirement specification has been fulfilled, as a way of trying to find bugs in the system, or to being able to make sure that new or changed functionality has not introduced bugs into the system.

3.5.1 Test automation

Test automation is a technique in which replicable test cases are written, set to run automatically and compared to expected results [54]. While taking longer than informal testing to setup, the replicability of informal tests are hard, while they are trivial for automated tests already written. This consistent behaviour is desirable to make sure that new bugs are not introduced when functionality is changed or as a way of ensuring that the system behaves as specified. The downside of using test automation lies in its inability of discovering bugs which depend on specific circumstances. Since it may depend on a test writer knowing that a specific circumstance may lead to an error, which is often not the case.

3.5.2 Load testing

Load testing is a form of test automation in which a system is put under heavy usage from automated queries being sent [34]. This is done to simulate the behaviour and response of the system during heavy usage from real users. For a multi-user system this can be done by instantiating a multitude of virtual users that attempts to use the system. By using load testing with logs the developer is able to see things such as, the response time, bottlenecks and maximum capacity to name a few things.

There are many different tools for load testing, two of the more commonly used open-source load testing tools are: Tsung and Apache JMeter.

Tsung

Tsung, previously known as IDX-Tsunami, is a load testing tool developed in Erlang. Erlang was designed and created by Ericsson, its an open-source language which was designed for the building of robust fault-tolerant distributed applications [38]. Tsung can stress HTTP, Simple Object Access Protocol (SOAP), PostgreSQL and MySQL servers among others. After the release of version 1.3.2 Tsung supports a limited subset of JSONPath natively [37]. Tsung does not provide any Graphical User Interface (GUI), the test sessions and cases are written in Extensible Markup Language (XML) and the statistic reports are generated by default as HTML documents.
Apache JMeter
Apache JMeter is a load testing tool, developed and maintained by Apache Software Foundation. The tool is written in Java and comes with a GUI for creating and running test cases. JMeter has the ability to stress many different protocols/server types, among them are HTTP, File Transfer Protocol (FTP), but also DBMSs through Java Database Connectivity (JDBC) [22]. The tool has a highly extensible core, allowing developers to easily extend the functionality of the tool. JMeter does not have native support for either JavaScript Object Notation (JSON) or PostgreSQL directly, but this functionality can be supported through extensions.

3.5.3 The importance of load testing web applications
The Quality of Service (QoS) of a web application is often measured in terms of response time, throughput, and availability [25, 34]. QoS management is an important part of a web application, controlling the quality of a service will result in a quality product and satisfy the users expectations [7]. A poor QoS will result in a decreased user base of the system, the users gets frustrated if the service is slow or unavailable. If the user amount drops, some business opportunities are lost. Therefore it is important to load test the system to see where potential bottlenecks are, where the system needs an upgrade, or which components needs to be optimized to handle heavy load [3]. Finding these bottlenecks within a system can save both time and money for the hosts of the web application. The resources can be spent on improving and upgrading the weaker parts of the system to reduce delays and increase user experience.

A web application can evolve over time, adding new functionality or changing the IT infrastructure of the system. It is therefore important to load test the system on a regular basis, to detect if the changes has introduced any potential performance problems, then the problems can be fixed before the clients experiences them [34].

3.6 Game design
For many game applications there exists concepts such as, payments in game and social sharing that are important for the games and their design. While these concepts are of less consequence for the work done in the thesis, they put the thesis in a context of where it will be used.

3.6.1 Payments in games
Payments in game as a paradigm started to emerge in the mid to late 2000s. The objective of payments in game is often to place a user in a pleasant and enjoyable gaming experience before monetizing it [13], making use of the initial lack of payments to accumulate a huge user base. This often leads to two types of players playing the game, the players that plays the game for free, which often represents 90-99%, and the paying players representing the rest [12]. Of these paying players, there exists three types: the minnows, dolphins, and whales each quantifying the small, medium and high paying players. For the game to remain profitable a small amount of whales, a large amount of both minnows and dolphins, or a combination of whales, minnows and dolphins is needed.

There exists a strong polarized view on the idea of payments in-game. With developers and publishers arguing that allowing players the ability to play the game free of charge, allows the players to test if they enjoy the game and want to spend time and money on it. This can then be decided on before any such commitment has already been made [1]. They also argue that this business model allows the developers to continually develop and improve the game.
With the player community arguing that what is called dark patterns has come from this. Where developers deliberately implement roadblocks and tedious elements further inside the game, that the player will not notice until after the player has committed to the game. These elements can then only be cleared through in game purchases which hinder the game flow [59]. There is also the argument that it becomes an ethical dilemma when young players are able to make purchases so easily, which has come to light recently.

What has come from this is that a middle ground is needed, where in-game payments exists, but does not hinder the players progression. Also there should exist a technological solution where young players are unable to make in-game purchases without a parent’s consent.

3.6.2 Viral marketing

Viral marketing or virality can be defined as the tendency of some online content (e.g. an image, article, video, app or service) to be circulated rapidly and widely between internet users. Continually spread by its readers, viewers or users to entice others to experience the content.

Two concepts of virality is word-of-mouth and viral marketing [32].

Viral marketing can be seen as the use of viral sharing for marketing purposes and Helm [28] defines the term in the following way:

"Viral marketing can be understood as a communication and distribution concept that relies on customers to transmit digital products via electronic mail to other potential customers in their social sphere and to animate these contacts to also transmit the products."

The term was introduced at first in an article written by Jeffrey Rayport at Harvard Business School in 1996, the article was called "The Virus of Marketing" and it was published in the business magazine Fast Company.

Viral marketing is sometimes also called electronic word-of-mouth. The term word-of-mouth can be described as the sharing of information (a product, promotion, etc.) with a friend, family, colleague or a person with similar interests. Word of mouth has been an interesting topic discussed by marketing researchers in more than five decades [32].

Why do people share viral content?

Emotions play a huge role when talking about why people decide to share viral content. If a viral marketing message builds an emotional connection with a person, that person is more likely to spread the message. There are six primary emotions that can have an impact on people wanting to share viral content, the emotions are: surprise, joy, sadness, anger, fear, and disgust. Each feeling can have different impact on different persons.

Positive content is more likely to be shared rather than negative content, but it has been shown to be a bit more complex than that. Content that evoke high-arousal emotions tend to be more viral, independent if the emotions are positive (i.e. joy) or negative (i.e. anger). If the content evoke a more deactivating emotion (i.e. sadness) it is less likely to be shared by the recipient [5]. Viral marketing messages that evokes surprise is only effective when it also evokes another primary emotion (i.e. both surprise and joy resulting in delight or surprise and disgust resulting in humor) [17].

Building an emotional connection might not be enough for viral success. If a viral marketing message is cleverly targeted to a group of people, it will increase the chances of being spread rather than just randomly select people.

An example of using a targeted group is when Motorola had a viral marketing campaign for the V70 model. Motorola used people who had previously registered on their website, since these people had shown interest in Motorola products and were a perfect group to target.
for their viral campaign. In two weeks after the campaign started the database had increased with 400%. On average 75% of the recipients in the viral campaign forwarded the message to at least one more person, 40% of the people who got the message went to Motorola’s website and wanted to know more about the new model [17].

Viral marketing in games

Viral marketing can be used in games in a way to let the existing users spread the game to other people, to recruit new players to the game and increase the user base. Four sharing mechanism characteristics for products are unsolicited messages, messages with incentives, direct messages from friends and broadcast messages from strangers [49]. These different mechanism for sharing have different impact on the success of the product.

Unsolicited messages Messages are considered unsolicited if the receiver of the message has not expressed interest in receiving the message. This sharing mechanism have a significant negative effect for viral marketing.

Messages with incentives Messages with incentives provide benefits for the receiver if the receiver decides to use the product compared to regular consumers. Incentives has a positive effect for viral marketing in games, but a negative effect for utilitarian products.

Direct messages from friends This mechanism is the most common one for games where a person can send a recommendation of a product directly to a friend. Direct messages from friends can have different effects for the products success whether it is a game or a utilitarian product. For utilitarian products it is shown that this mechanism have positive effect, but for games it can have both positive and negative effects.

Broadcast messages from strangers Broadcast messages from strangers is a mechanism that allow a user to share the product with a lot of people that has no relationship to the sharer. Broadcast messages received from strangers has the most negative impact for the products success if the product is a game while it has really positive effect if it is a utilitarian product.

3.7 Research methodologies

A case study is an empirical method used for investigating contemporary phenomena in their context. There are three other major research methodologies which are related to case studies: survey, experiment, and action research. Survey is a collection of information from a specific population. Experiments are conducted by measuring the effects of one variable when changing another variable. Action research aims to influence or change some focused aspect in a research. The difference between a action research and a case study is that action study is involved in the change process, while a case study is purely observational [46].

3.7.1 Case study

Conducting a case study consists of five major process steps: case study design, preparation for data collection, collecting evidence, analysis of collected data, and reporting. Since case study methodology is a flexible design strategy, the process steps when conducting the research are often iterated through several times [46]. The purpose of conducting a case study can vary and different methodologies serve different purposes. Four types of purposes for conducting a case study [46]:

• Exploratory, investigating what is happening or to find ideas for new research.
• Descriptive, describing a situation or phenomenon.
• Explanatory, searching for an explanation of a problem or a situation.
• Improving, trying to improve some aspect of the studied phenomenon.

3.7.2 Agile methodology
Agile methodologies welcome change and unpredictability in software projects [51]. These methodologies rely on the creativity of the people in the project rather than on processes to deal with unpredictability [10]. Therefore these methodologies are usually addressed toward small- to medium-sized teams developing software with often changed or vaguely defined requirements [40]. For the adjustment of changes in the requirements, agile approaches are recommended to use short iterations for the feedback loop with customers and management. These iterations are recommended to be no longer than six weeks [30]. There are many different agile approaches out there, Extreme Programming, Scrum, Crystal, to name a few. Agile approaches combine these short iterations with feature planning and dynamic prioritization. Dynamic prioritization are when the developers uses the end of an iteration to reprioritize the features that the customers or management wants done in the next cycle. This can lead to adding new features and discarding planned ones [30].

One benefit of using an agile methodology in a project is that the customer can quickly see working code. A drawback of this is the lack of documentation, since agile approaches discourage documentation beyond the code [36]. Because of this, much of the knowledge resides in the heads of the members in the development team, this results in members being less interchangeable and this can have consequences for how the projects are managed [18]. Another drawback is that if the customers do not have a good sense of direction, the result of the product will suffer [30].

3.7.3 Requirement elicitation
Requirement elicitation is the process of identifying and elaborating requirements for a system. This is a complex process that begins in an early stage when developing a system and continues throughout the project. An important part of eliciting requirements are to uncover and extract what the potential stakeholders wants [60]. There are many different techniques and approaches that can be used for eliciting requirements. Interviews are one of the most commonly used techniques, it is an effective technique that allows mistakes and misunderstandings to be cleared up during the interview [39]. The results from an interview can vary a lot depending on the interviewers skill to uncover and extract information. There are three different types of interviews: structured, unstructured, and semi-structured [60].

Structured interviews are when the interviewer has prepared a set of questions to retrieve specific information from the interviewee. This type requires for the interviewer to be well prepared. It is important to know beforehand which are the most appropriate questions to ask and when to ask them, to gain as much information from the interview as possible. One drawback with this type of interview is that they tend to limit the investigation of new ideas [60]. Unstructured interviews are when an open discussion is held rather than using predetermined questions. Unstructured interviews are useful when exploring the domain and to gain an understanding of the problem. This technique can often be used as a precursor to other techniques, such as structured interviews. There is a risk with unstructured interviews that the discussion can easily put too much focus on some specific part of a system and miss another part completely. A semi-structured interview is a combination of a structured and a unstructured interview [60].
Chapter 4
Method

This chapter describes the approach for designing, implementing and testing the system for this thesis.

4.1 Agile development

An agile process was used in the development of the product. Working in an agile software development process requires the developer to be flexible, since requirements can change, be removed or new requirements can be added during the development of the product.

4.1.1 Iterative development

During the design and implementation phase of this thesis, an iterative development process were used. Small parts of the functionality was created or changed, tested, and then fixed if a problem existed. Iterating on previous work, extending the functionality of the system, bit by bit. Meetings was held once a week with the stakeholder during 14 weeks to give an update of how the implementation was going, which obstacles had occurred, and what was next to implement.

4.1.2 Requirement elicitation

To define the base requirements for the project, an unstructured interview were held with the stakeholder on the first week of the design and implementation phase. During this phase, additional unstructured interviews were used on some of the weekly meetings with the stakeholder. This was done to look over and discuss a subset of the requirements, to see if any changes were to be made.

4.2 System design

After the initial requirements were elicited, the design approach were discussed. Because an agile development process was used, the system design had to be flexible to accommodate possible changes in the requirements. Since the system were to follow the REST architectural style, the constraints defined by REST had to be taken into consideration when designing the system.

During the designing of the system, three major aspects were of interest: the design of the database and access to it, complying with REST constraints when designing the API, and security considerations.
4.2. System design

4.2.1 Designing the database

Based on the initial requirements, the schema for the database was defined. This was done by finding common denominators in the requirements and describing these as objects when appropriate. These objects where then defined with attributes, describing the object, and with relations in regard to each other. When defining the schema, SQLAlchemy was used to build the objects and create the relationships between them. SQLAlchemy then used the schema to generate the tables for storing the defined objects. When the requirements changed during the design and implementation phase, the database schema changed accordingly, which lead to the tables being re-generated by SQLAlchemy.

4.2.2 Designing the application programming interface

Designing the API using an agile development process requires a simple structure for accessing and modifying data, since the models of the system can change at any time. With the REST constraints in mind when designing the API, each resource was to be given an unique URI for accessing it. The URI for the resources is decided using path variables as a way of encoding hierarchy between the resources. A root URI using the string /users/, would return a list of all users. Using the sub path /users/<username>, would return information about the specified user, who is part of a subset of all the users in the system. The path variables was decided, so that it represented the resource it linked to, and so that it was human readable and understandable.

To fulfill statelessness for the system, no client context was chosen to be stored on the server between the clients requests. The clients request had to contain all the state specific information needed in the request. Previous states and resources visited by the client was not saved on the server. Instead the client had to store and send information gathered from previous requests, if a request was to depend upon previous actions.

Since every system resource was to be uniquely addressable, the response from a HTTP GET request is cacheable. This allows common request responses to be stored in caches, reducing the load on the systems server which increases performance and scalability. Every request made by a client is responded by the system as a JSON response and with a HTTP status code. Using this approach a client can, based on the status code, know if the request was successful and act upon the response thereafter. If the status code implies an error occurred somewhere, the status code in combination with the response message will give the client information about what type of error occurred.

By designing the API this way it can be considered generic, since the system becomes accessible by any client able to send and receive standard HTTP requests.

4.2.3 Security

Because of the need for storing sensitive user information and since communication with clients outside of the system is necessary, a secure mindset was needed. To improve the integrity and confidentiality of the user data stored, passwords are obfuscated using a hashing algorithm and then stored, instead of being stored in plain text. As a way of improving confidentiality and integrity in the communication with clients, communication is done using TLS. To improve the confidentiality and allow for a higher data integrity, users that log on to the system are generated and sent an authentication token. This is done to ensure that some actions can only be done by users holding the token. This is also done as a way of ensuring statelessness on the server, since the server only need to check if the token is valid. Since only the authenticated user has access to a token, an valid token proves that the user is authenticated to do the requested action. To improve the integrity of individual users sessions, usage of CSRF protection through tokens was used.
When retrieving data from the system, every HTTP GET request is considered safe because there are no confidential data that can be retrieved. While for every state-changing request received on the system, authentication is required to enhance the integrity of the system resources. This works in combination with all GET requests being cacheable, since no authentication is needed to access the response from accessing a resource using a GET, the response can be safely stored in a cache.

4.3 Implementation

During the implementation there was two phases, in which focus was on different subjects. In the first phase, focus was on users and user relationships such as clans and highscores. In the second phase, focus was on the store and payment service. During both phases, client form data were validated before executing any business logic. To validate the form data, the Python package WTForms were used, which also includes tools to help protect against CSRF [52]. By putting business logic behind the data validation, access attempts to the database is kept to a minimum.

4.3.1 Users and user relationships

During the initial implementation phase the models for: users, clans, highscores, and their relationships were implemented. These relationships were of the types: friends, clan members, and user accounts. There are two different user account types; parent accounts for adults and children accounts which are mapped to a parent account. The implementation for the models and relationships were done using the classes defined by SQLAlchemy. These classes allows the models to define the columns, column type, and constraints that the model adheres to. With each model working as a façade for the persistent data stored in the database, functionality to change the data is done through instances of the data model.

The system was then implemented so that requests received on the system follows the uniform interface constraint and is sent to the correct part of the business logic, based on the URI. The different operations are done based on the method header in the HTTP request. These operations where implemented to be small in scope and to filter the request based on sent data, if the referenced resources existed, and if the user was authorized to make these changes if applicable. If there was a failure in any of the filters, an error message was sent as a response together with an appropriate status code. If no error occurred, the operation was to be performed and then a response message was sent back, also here with an appropriate status code. These operations were to make use of instances of the resources referenced and accessed through the resources models façade.

4.3.2 Store and payment service

Before implementing the payment service, the models for the store, virtual items, physical items, orders, gifts, and the relationships between these were created. The store was used for fetching all the items and charging customers when purchasing both physical and virtual items. The payment service provider Stripe was implemented in the store for making the payment charges. The system also checked with Stripe whether the payment was valid before granting the items to the customers, if it did, Stripe sent a receipt to the customers supplied email address. Stripe handles the credit card information, while none of the supplied payment information were stored in the system.

If a customer was about to purchase a virtual item, two different purchase choices were implemented: buying the item as a gift or buying it to oneself. If the item was chosen to be purchased as a gift, the email address supplied by the customer received a gift code. This gift code were then added in the database and could be activated by any user to receive
the virtual item. If the purchase was to oneself, the users possessions in the database were updated accordingly.

4.4 Load testing

After the implementation of the system was done, focus was shifted to designing the test cases for load testing the system. These test cases was designed to be used by Tsung as a test suite to measure the impact of an increasing amount of concurrent users. While designing the test cases, emphasise was on the variety and quantity of the tests, rather than on system code coverage. Since the purpose of the load testing of the system was to simulate heavy usage, the test cases were written to be small in scope and to be able to be instantiated and run quickly.

This lead to the introduction of an exception in the system, whereupon an authentication token using the string testToken was accepted as valid. In some sessions with lower probability true authentication was used anyway, to simulate a more realistic usage of the system. The reason for this exception was that the expected realistic usage of the system, was that the majority of the requests are sent from users who are already logged in. Requests based on creating a new user account or logging in to an existing account are expected to be in a minority. But since authentication is needed for certain actions, and the authentication token can not be stored between the sessions easily in Tsung, a concession was made. This was because the hashing algorithm for obfuscating the newly generated users password, or comparing it to the one supplied during an authentication attempt, was extremely resource intensive compared to all other operations.

The data collected when running the tests was then used to generate a statistical report with a script supplied by Tsung. These reports included graphs and tables summarizing the systems behavior and response time for the different test cases. The statistics from the reports could then be compared to other reports with more, or less, concurrent users.
Chapter 5
Results

This chapter describes the final results from the design, implementation, and testing of the system in this thesis.

5.1 Requirement elicitation

The first unstructured interview with the stakeholder resulted in a set of requirements for the project:

- A user should be able to create a new account.
- A user should be able to create a child account.
- A user should be able to search for other users.
- A user should be able to follow other users.
- A user should be able to unfollow followed users.
- A user should be able to save highscores.
- A user should be able to see his previous highscores.
- A user should be able to see the latest highscore of followed users.
- A user should be able to see the highscores for a specific level
- A user should be able to share the game to others by email.
- A user should be able to create a clan.
- A user should be able to invite others users to join its clan.
- A user should be able to accept/reject a clan request.
- A user should be able to search for clans.
- A user should be able to participate in timed events.
- A user should receive a virtual gift after completing a timed event.
- A user should be able to buy items as gifts.
- A user should be able buy items from an in-game store.
- A user should be able to see if items are on sale.
- A user should be able to buy items using in-game currency.
During implementation of the system, the weekly meetings with the stakeholder resulted in some changes to the requirements. The following requirements changed:

- A user should be able to follow other users.
- A user should be able to unfollow followed users.
- A user should be able to see the latest highscore of followed users.

The requirements were changed to:

- A user should be able to send friend request to other users.
- A user should be able to accept/reject a friend request.
- A user should be able to remove a friend.
- A user should be able to see the latest highscore of the users friends.

The following requirements were also added:

- A user should be able to send a request to join a clan.
- A user should be able to leave a clan.
- A user should be able to reset password.
- A user should be able to change password.
- A user should be able to place and pay for an order of physical goods.

The following requirements were removed:

- A user should be able to buy items using in-game currency.
- A user should be able to participate in timed events.
- A user should receive a virtual gift after completing a timed event.

A summary of all the final requirements can be seen in Appendix A.

5.2 System design

The result of the system design was a three-layered system.

As can be seen in figure 5.1, the first layer filters the requests based on the URI and method header. If the URI corresponds to a predefined route, the request is sent to the next layer. If no such route exists, or the method header for that route is not defined, an error message with corresponding status code is sent back to the client.

In the second layer, a clients request is put through filters, validating the data in the request. In all resource state-changing requests, both CSRF and user authentication tokens are also required. The exempt to these requests are creating a new user and signing in as a user. In these requests only the CSRF token is required, and authentication is done through a password system. If the request is valid, instances of the referred resources are retrieved from the third layer. These instances are then used to handle the clients request for retrieving, creating, modifying or deleting resources. A response is then created containing response data and a HTTP status code, based on the operation used. If an error occurred during validation, the business logic for handling the request aborts, and an error message is sent back with a corresponding status code.

The third layer works as a façade for the persistent data stored in the database. This layer contains all the logic and queries for accessing and modifying the data in the database.
5.2. System design

5.2.1 Database

Figure 5.2 shows every entity and every relationship in the database, the entities and all the many-to-many relationships are tables containing data. The entities also shows which attributes they include, while the many-to-many relationships includes foreign keys referencing the entities.

One thing of notice in the database, is the inheritance between the entity Items and its sub-classes. The entities Game Items and Physical Items both inherits from Items, this allows for all items to have both a name and description. The store in the system uses the entity Item Sales to set the prices of all items and setting discounts for specified time intervals.

The entity Users have multiple many-to-many relationships with itself to handle friends, friend requests, and connecting parent and child accounts. It also has multiple many-to-many relationships with the entity Clans to handle clan invites, clan members, and also a one-to-many relationship to connect a clan leader to a user. The Users entity also ended up with a many-to-many relationship with the entity Game Items, to see which virtual items a user has acquired.

When creating a physical order, some personal data had to be stored. This was for knowing where to send the purchased objects. The order also contains a shipping status which is updated depending on whether an order has been created, payed for, or sent.
Figure 5.2: EER diagram over the database
### 5.2. System design

#### 5.2.2 Application programming interface

Tables 5.1, 5.2, 5.3, and 5.4 show the public APIs accessible through the web. They represent all the actions that can be taken on the system by a client.

Table 5.1: A table showing the routes and HTTP method applicable for the user resource

<table>
<thead>
<tr>
<th>Route</th>
<th>HTTP method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/users/&lt;username&gt;</td>
<td>GET</td>
<td>Retrieves user information</td>
</tr>
<tr>
<td></td>
<td>PUT</td>
<td>Create a new user, or modify an existing user</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/possessions</td>
<td>GET</td>
<td>Retrieves a list of all the users possessions</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Activates a gift code to receive an item</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/possessions/&lt;gameName&gt;</td>
<td>GET</td>
<td>Fetches the specified users possessions for a certain game</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/friends</td>
<td>GET</td>
<td>Fetches a users friends</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Send a friend request to a user</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Reject a friend request</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/friends/&lt;friend&gt;</td>
<td>GET</td>
<td>Get the user information from a specific friend</td>
</tr>
<tr>
<td></td>
<td>PUT</td>
<td>Accept a friend request from a specific user</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Remove the specified friend</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/clans</td>
<td>GET</td>
<td>Fetches all the clans that a user is member in</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Send an invitation to a user to join a clan</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Route</th>
<th>HTTP method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/users/&lt;username&gt;/clans/&lt;clanName&gt;</td>
<td>GET</td>
<td>Fetches the clan information for a specific clan that a user is member in</td>
</tr>
<tr>
<td></td>
<td>PUT</td>
<td>Accept a clan invitation</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Reject a clan invitation</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/parent</td>
<td>GET</td>
<td>Get user information from a users parent</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/children</td>
<td>GET</td>
<td>Fetches user information about a users children</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Create a new child account connected to a specific user</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/children/&lt;child&gt;</td>
<td>GET</td>
<td>Fetches the user information from a specific child account</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/signOut</td>
<td>POST</td>
<td>Sign out the logged in user</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/signIn</td>
<td>POST</td>
<td>Sign in a specific user</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/forgotPassword</td>
<td>POST</td>
<td>Request a code to change the forgotten password</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/recreateNewPassword</td>
<td>POST</td>
<td>Create a new password for a user with a forgotten password code</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/refreshToken</td>
<td>POST</td>
<td>Refreshes the authentication token</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/&lt;gameName&gt;/highscores</td>
<td>GET</td>
<td>Fetches all the highscores made by a specific user for a certain game</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/&lt;gameName&gt;/highscores/&lt;levelName&gt;</td>
<td>GET</td>
<td>Fetches all the highscores made by a specific user for a certain game and level</td>
</tr>
<tr>
<td></td>
<td>PUT</td>
<td>Save a new highscore for a specific game and level</td>
</tr>
<tr>
<td>/users/&lt;username&gt;/&lt;gameName&gt;/promote</td>
<td>POST</td>
<td>Sends an email promotion of the game to a specified email address</td>
</tr>
</tbody>
</table>
### Table 5.2: A table showing the routes and HTTP method applicable for misc. resources

<table>
<thead>
<tr>
<th>Route</th>
<th>HTTP method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/games/&lt;gameName&gt;/highscores/&lt;levelName&gt;</td>
<td>GET</td>
<td>Retrieves the top highscores for the specified level in the game</td>
</tr>
<tr>
<td>/getToken</td>
<td>GET</td>
<td>Retrieves a CSRF token</td>
</tr>
</tbody>
</table>

### Table 5.3: A table showing the routes and HTTP method applicable for the clan resource

<table>
<thead>
<tr>
<th>Route</th>
<th>HTTP method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/games/&lt;gameName&gt;/clans</td>
<td>GET</td>
<td>Retrieves a list of all the clans in the game</td>
</tr>
<tr>
<td>/games/&lt;gameName&gt;/clans/&lt;clanName&gt;</td>
<td>GET</td>
<td>Retrieves information about the specified clan in the game</td>
</tr>
<tr>
<td></td>
<td>PUT</td>
<td>Used to create a new clan in the specified game</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Deletes the specified clan</td>
</tr>
<tr>
<td>/games/&lt;gameName&gt;/clans/&lt;clanName&gt;/members</td>
<td>GET</td>
<td>Retrieves a list of all the members in the clan</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Used by a user when sending a request to join the clan</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Used to reject a users request to join the clan</td>
</tr>
<tr>
<td>/games/&lt;gameName&gt;/clans/&lt;clanName&gt;/members/&lt;memberName&gt;</td>
<td>PUT</td>
<td>Used to accept a users request to join the clan</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Used to leave or kick a user from the clan</td>
</tr>
</tbody>
</table>
5.3 Implementation

SQLAlchemy was chosen as the DBMS for the system based on the mapping functionality between Python objects, its enterprise-level persistence patterns, and a discussion with the stakeholder. Creating the database models was done using SQLAlchemy's predefined Model class. In this class the database tables name and attributes with constraints for the model is defined. The code in figure 5.3 shows how the Item model is defined.

```python
class Item(db.Model, DatabaseObject):
    __tablename__ = 'items'
    id = Column(db.Integer, primary_key=True)
    name = Column(db.String(50), unique=True, nullable=False)
    description = Column(db.String(1000))
```

Figure 5.3: A code snippet creating a database model

Figure 5.4 shows how routes are defined in the system. It also shows which HTTP methods are defined for that route and how the request is passed along to the next system layer. Note that @userRoutes uses the URI prefix /users to ensure that all requests are sent to the user routes.

Table 5.4: A table showing the routes and HTTP method applicable for the store resource

<table>
<thead>
<tr>
<th>Route</th>
<th>HTTP method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/store/games/&lt;gameName&gt;/items</td>
<td>GET</td>
<td>Retrieves a list of all the items in the specified game</td>
</tr>
<tr>
<td>/store/physicalItems</td>
<td>GET</td>
<td>Retrieves a list of all the physical goods</td>
</tr>
<tr>
<td>/store/games/&lt;gameName&gt;/items/&lt;itemId&gt;</td>
<td>GET</td>
<td>Retrieves specific information about the item</td>
</tr>
<tr>
<td>/store/physicalItems/&lt;itemId&gt;</td>
<td>GET</td>
<td>Retrieves specific information about the item</td>
</tr>
<tr>
<td>/store/items/&lt;itemId&gt;/paymentInformation</td>
<td>GET</td>
<td>Retrieves payment information about the specified item</td>
</tr>
<tr>
<td>/store/items/&lt;itemId&gt;/buy</td>
<td>POST</td>
<td>Used to buy the item</td>
</tr>
<tr>
<td>/store/placeOrder</td>
<td>POST</td>
<td>Used to place an order for a set of physical goods</td>
</tr>
<tr>
<td>/store/placedOrders/paymentInformation/&lt;orderId&gt;</td>
<td>GET</td>
<td>Retrieves payment information about the placed order</td>
</tr>
<tr>
<td>/store/placedOrders/pay</td>
<td>POST</td>
<td>Used to pay for the placed order</td>
</tr>
</tbody>
</table>
5.4 Load testing

Figure 5.4: A code snippet showing how routing is done

```python
@userRoutes.route('/<username>', methods=['GET', 'PUT'])
def userIndex(username):
    if request.method == 'GET':
        return usrCtrl.getUserInformation(username)
    elif request.method == 'PUT':
        return usrCtrl.createOrChangeUser(username, request)
```

Figure 5.5: A code snippet showing how information about a user is sent

```python
def getUserInformation(username):
    usr = User.getUser(username)
    if usr is None:
        return jsonify(error='User does not exist'), 404
    c = []
    append = c.append
    for clan in usr.getAllClans():
        append(clan.clanName)
    return jsonify(username=usr.username,
                    nationality=usr.nationality,
                    clans=c), 200
```

Figure 5.6: A code snippet showing how form data is validated

```python
class SaveUserHighscoreForm(Form):
    score = IntegerField('Score', [validators.InputRequired(),
                                   validators.NumberRange(min=0)])
```

5.4 Load testing

The different sessions used during load testing can be seen in Appendix B. As can be seen in
the different sessions, each one has different weights describing the probability of a session
being run. This was done to try and simulate a more realistic usage, with different users,
doing different actions. The testing also uses some predefined data inserted into the system
before each test case was run. The reason for this was to avoid that every session had to create
a new user account to test the system.

The tests were divided into three phases, each phase had different user arrival rate. The
first phase when testing the system on OpenShift had an arrival rate of three users per second,
the second phase had four and the last phase had five users per second. During the testing of
the system hosted on the OpenShift server at Linköping University, a major problem arose.
The system behaved inconsistent between each test case, even if the same test case was run
twice. As can be seen in figure 5.7 the request duration time from running the same test case
twice, was very inconsistent.
5.4. Load testing

(a) OpenShift test one

(b) OpenShift test two

Figure 5.7: Test output from testing the system on OpenShift using the test case described in Appendix C.1

Because of the inconsistent behaviour, a decision was made to host and test the system on a local machine. This local machine was a Packard Bell TJ75, this would represent a lower bound for the hardware that the system can be expected to be hosted on.

After moving the system to the local machine, the arrival rate of the different phases of the tests were changed. The first phase was increased to 10, the second phase to 11 and the third phase to 12 users per second. The results from the first test run on the local machine can be seen in figures 5.8a, 5.8c and 5.8e. The results from the first test shows a spike at the end of the request duration graph. To see how the system behaves during a longer test session, a second test was run. The results from the longer test can be seen in figures 5.8b, 5.8d and 5.8f. The results from figure 5.8f shows that the system can not handle the large amount of simultaneous users during the third phase of the test. This can also be seen in figure 5.8b where the response time for each request has increased significantly. Since the system is unable to handle the massive load, the request rate drops.
5.4. Load testing

(a) Mean request duration on local test one

(b) Mean request duration on local test two

(c) Request and connection rate on local test one

(d) Request and connection rate on local test two

(e) Simultaneous users on local test one

(f) Simultaneous users on local test two

Figure 5.8: Test data from running two different test cases on the localhost with three phases each. With the pictures to the left representing data from the test case described in Appendix C.2 with each phase being 400 seconds long. The pictures to the right representing the test case described in Appendix C.3 with each phase being 600 seconds long.

To see if the system behaves consistently with the same arrival rate, the three phases were extended to six phases with the last three mirroring the first three phases. An extended duration was set for phase three to let the system finish all previous requests before starting phase four. Figure 5.9a, 5.9c and 5.9e shows the results of the first test case with six phases. Figure 5.9b, 5.9d and 5.9f shows the results of a similar but longer test case. Both test cases points to a similar behavior of the system during the testing when using six test phases.
5.4. Load testing

(a) Mean request duration on local test three  
(b) Mean request duration on local test four

(c) Request and connection rate on local test three  
(d) Request and connection rate on local test four

(e) Simultaneous users on local test three  
(f) Simultaneous users on local test four

Figure 5.9: Test data from running two different test cases on the localhost with six phases. The third phase is twice as long as the rest of the phases to accommodate a cooldown time. With the pictures to the left representing data from the test case described in Appendix C.4 with each phase being 120 seconds long. The pictures to the right representing the test case described in Appendix C.5 with each test case being 240 seconds long.
Chapter 6
Discussion

This chapter contains a discussion about the results from this thesis, the method used, and the work in a wider context.

6.1 Results

Here we will discuss the results from the work done, putting it in context, and trying to explain what the results mean.

6.1.1 System design

By designing the database with the usage of SQLAlchemy, changes to a model were quick to introduce into the system. This meant that the design of the database were quick to adapt to changes in the requirements, if the system were to be refactored because of found flaws, or because of changes in the design or logic, SQLAlchemy simplifies the modification of the database. Which is one of the strengths of using SQLAlchemy for describing the relation between database content and application logic. Also because of the mapping between the Python objects and the data stored by the DBMS, an object-oriented approach to the data could be utilized [11].

One of the strengths in regard for this thesis with designing the system to be RESTful is the statelessness it supplies. By designing the system to be stateless the ease of horizontal scaling is increased, or as Leonard et al [44] puts it:

"This makes it trivial to scale your application up. If one server can't handle all the requests, just set up a load balancer and make a second server handle half the requests. Which half? It doesn't matter, because every request is self-contained. You can assign requests to servers randomly, or with a simple round-robin algorithm. If two servers can't handle all the requests, you add a third server, ad infinitum. If one server goes down, the others automatically take over for it. When your application is stateless, you don't need to coordinate activity between servers, sharing memory or creating "server affinity" to make sure the same server handles every request in a "session." You can throw web servers at the problem until the bottleneck becomes access to your re-source state. Then you have to get into database replication, mirroring, or whatever strategy is most appropriate for the way you've chosen to store your resource state."

Based on the results regarding logic flaws in payments found by Wang et al [56], none of the flaws found by them was found by us in the system. This is not to say that no logic flaws, other than those described by them, regarding the payment aspect can exist in the system developed. But based on their findings and how the system is designed, no flaw of the types found by them should exist in the system. The filtering of user form data, before any operations are performed in the system, works well as a way of ensuring that secure by default is used as a security concept. It also works well as an optimization approach, since
less resources are used by the system for requests, which lack certain data, or is faulty in some other way.

6.1.2 Load testing

When the system was hosted on the OpenShift server at Linköping University, no knowledge about the server, its hardware, and its load from other applications were known. Because of this, conclusions about the results were difficult to make. This also may have been a big factor in the inconsistencies found in the result data from the test cases. So to estimate a expected lower bound for the systems performance during heavy load in a cloud-based solution, the system was hosted locally on non-dedicated hardware. A problem with hosting the server on the local system was that the functionality for storing content in caches between the test machine and the server was impossible. This was since the machine hosting the server, was the same machine running the tests. Instead the possibility of caching the content is not proven experimentally, but by concept only.

As shown in figures 5.8 and 5.9, the lower bound was with a concurrent user count slightly below 200 users, and with a request rate slightly below 40 request per second. The initial tests also pointed to that the server needs some time to recover from an overload of users requests. This recovery rate is proportional to how high the overload is, otherwise the server will not be able to handle any new coming requests. The problem with these results, is that there exists a difficulty of putting them in a context. While a result for the thesis question is found for this specific hardware, it lacks context to compare it to. Because of this lack of context, the QoS for the product can not be stated to be of good or bad quality, since these have not been enough specified. This, as Jorge et al [7] points out is important to satisfy a users expectations.

All the load tests in this thesis were run using Tsung as a testing tool. Tsung was chosen because the simple and easy setup, its ability to work out of the box, and the native support for JSONPath [37]. JMeter was another considered testing tool, but the lack of native support for JSONPath pushed Tsung as a stronger candidate to use as a testing tool. Because of time constraints only one testing tool was used, but the expected results should be about the same from using either tools [53].

6.2 Method

Here is where the methods applied in this thesis is discussed, both what worked, and how it could have been improved.

6.2.1 Agile Development

One thing we would have put more emphasise on if we were to redo this thesis, is to specify the requirements to a greater extent and try to be more specific regarding functionality. One of the problems during this thesis was a lack of specification regarding functionality and expected behavior early on. This could have been achieved by being more thorough during the first unstructured interview with the stakeholder. Even though we used an agile development process and we knew that requirements could change, be added, or be removed during this thesis [51], we could have minimized the changes in the requirements by doing so. In effect, having to spend less time, changing existing functionality, and more time creating new functionality. But using short iterations between the meetings with the stakeholder allowed us to update the requirements frequently, to not waste too much time on features that were later removed or changed [30].
6.2.2 System design

The usage of a RESTful approach in combination of using SQLAlchemy, allowed for a more modularised approach regarding how functionality was written. By defining the URI path for individual resources and following the uniform interface, functionality for the resources was separated into small functions. This separation into small modules, allowed for quicker debugging of the code, and for a quicker process for developing smaller parts of the functionality wanted of the system. The usage of this separation into smaller modules, also allowed for faster refactoring of the code when the requirements were changed, since the code affected was generally of smaller scope. This meant that large parts of the code base could stay the same, and small parts could be changed and debugged, without impacting the overall code.

The design of the API, while specific for this case, follows the ideas put forth by Leonard et al [44], both regarding being uniquely addressable and the uniform interface. Because of this, while the path variable structure and naming is highly specific, the overall design and format follows their ideas.

6.2.3 Implementation

The separation of the implementation into the two major phases regarding user relationships and the payment service helped with the focus on functionality during this time. By focusing on smaller parts of the system, a more iterative approach to develop the functionality was used. Instead of having to balance the concerns for multiple contexts at once, modularity was favored, improving the quality and the decoupling of the system. As stated in 6.2.2, by designing the system to be modular, the newly changed or introduced functionality would not break older functionality, because of the decoupling in the system.

6.2.4 Load testing

If this thesis could be done with extra resources, testing the system on different hardware and with multiple servers accessed through load balancing would be of interest. This would be of interest in the regard to see how the systems load capacity could be increased using either vertical or horizontal scaling. Which as Jeremy et al tells [19] is important for the service to grow and gain a following. With the introduction of a horizontal scaling solution, the final design would have been different. Mostly by the introduction of a load balancer and with some changes of how the authentication of users are done. This would also allow for testing whether the caching of content works, and how effective this is at reducing the load for the individual servers. While vertical scaling would not impact the system design, the results could be more conclusive regarding effectiveness, if the system was to be compared between servers hosting the system, but with different hardware. This could also have lead to put more focus on where improvements in the system could be made, and how this would be facilitated.

The literature study showed that while there are not a lack of scientific literature regarding load testing, even though load testing as a concept is fairly new, it is mostly written about in abstract terms. Because the load test results from the literature’s systems is so application and hardware specific, comparisons between results becomes meaningless. This makes it so that the results from the literature is of almost no interest. In that no parallels can be made regarding if the results of a system, points to it being efficient or not.

6.3 The work in a wider context

An ethical concern that may exist about this thesis, would be regarding personal integrity. Since some personal information is needed to be stored in the system, this is never done without a users knowledge though, and can only be done if the user supplies the information.
Another concern can be regarding information supplied to the payment service. Payment information is not handled on the system, instead all payment information is handled by Stripe, therefore the user has to trust Stripe with the sensitive information. This is done to reduce the responsibility on our side regarding the handling of this information. To reduce the ethical concern regarding children doing purchases with real money, lead to the introduction of only parent accounts being able to make purchases within the system.

To try and improve the integrity and confidentiality of a users session and storage of data, common technologies such as TLS, CSRF, and hashed passwords are used at appropriate points in the system.
The purpose of this thesis was to design and implement a scalable back-end system that would supply payment functionality for web based game services. This purpose has been fulfilled to the extent that a system with payment functionality has been implemented. This system is accessible using standard HTTP requests irregardless of the platform used to send the requests.

Regarding the first research question, how a RESTful architecture can be implemented. One possible solution for implementing this was found following the method described in chapter 4 and the resulting architecture is the result of the work done in this thesis.

Regarding the second research question, how many concurrent users that can be serviced using this design within an acceptable response time. The load tests points to an increase of 11 users per second to a bound of 180 concurrent users, sending requests at the rate of 38 requests per second using the system hosted on the local machine. Anything more than this and the response time starts to increase and no guarantees can be made regarding fulfilling the acceptable response time.

By following the constraint defined by REST when developing a web service, scalability becomes a natural part of how one would design the system. The system in itself does not becomes scalable following these constraints, but the system is easier to design scalable.

7.1 Future Work

There are two main subject points that can build on the work done in this thesis. The first point would be to improve the scalability of the system. This could be done by implement horizontal scaling of the system. Introducing load balancers, redundancy of servers, and refactoring out authentication of users to a server handling only authentication. Another possible thing to extend the system with is adding more redundant databases and keep the data replicated between these databases. Improving the scalability could also be done by optimizing the code that the system is built on.

The second point would be to focus on the security of the system. While a secure state of mind has been kept during this thesis, the main focus of this thesis has not been the security aspect. This means that security vulnerabilities and weaknesses has not been tested for. This could be done by doing penetration testing on the system to find these potential flaws and mitigating these.
Appendix A
Requirements

- A user should be able to create a new account.
- A user should be able to create a child account.
- A user should be able to search for other users.
- A user should be able to send friend request to other users.
- A user should be able to accept/reject a friend request.
- A user should be able to remove a friend.
- A user should be able to save highscores.
- A user should be able to see his previous highscores.
- A user should be able to see the latest highscore of the users friends.
- A user should be able to see the top highscores for a specific level.
- A user should be able to share the game to others by email.
- A user should be able to create a clan.
- A user should be able to invite others users to join its clan.
- A user should be able to accept/reject a clan request.
- A user should be able to search for clans.
- A user should be able to buy items as gifts.
- A user should be able buy items from an in-game store.
- A user should be able to see if items are on sale.
- A user should be able to send a request to join a clan.
- A user should be able to leave a clan.
- A user should be able to reset password.
- A user should be able to change password.
- A user should be able to place and pay for an order of physical goods.
Appendix B
Tsung Test Suite

```xml
<sessions>
  <session name="http-create-user-test" weight="1" type="ts_http">
    <request subst="true">
      <match do="dump" when="nomatch">WILD8</match>
      <dyn_variable name="csrf_token" jsonpath="token" />
      <http url="/getToken" method="GET" version="1.1"></http>
    </request>
    <setdynvars sourcetype="random_string" length="25">
      <var name="rndstring1" />
    </setdynvars>
    <request subst="true">
      <dyn_variable name="auth_token" jsonpath="token" />
      <http url="/users/%rndstring1%/EGame/highscores/lavel1" contents="score=%random_score%" content_type="application/x-www-form-urlencoded" method="PUT" version="1.1">
        <http_header name="X-CSRF-TOKEN" value="%csrf_token%"/>
        <http_header name="Authorization" value="%auth_token%"/>
      </http>
    </request>
  </session>
  <session name="http-login-user-test" weight="1" type="ts_http">
    <request subst="true">
      <match do="dump" when="nomatch">WILD8</match>
      <dyn_variable name="csrf_token" jsonpath="token" />
      <http url="/getToken" method="GET" version="1.1"></http>
    </request>
    <setdynvars sourcetype="random_number" start="1" end="100000">
      <var name="random_score" />
    </setdynvars>
    <request subst="true">
      <http url="/users/%rndstring1%/EGame/highscores/lavel1" contents="score=%random_score%" content_type="application/x-www-form-urlencoded" method="PUT" version="1.1">
        <http_header name="X-CSRF-TOKEN" value="%csrf_token%"/>
        <http_header name="Authorization" value="%auth_token%"/>
      </http>
    </request>
  </session>
</sessions>
```
<session name="http-clan-test" weight="3" type="ts_http">
  <request subst="true">
    <match do="dump" when="nomatch">WILLBE</match>
    <dyn_variable name="csrf_token" jsonpath="token" />
    <http url="/getToken" method="GET" version="1.1"></http>
  </request>
</session>

<setdynvars sourcetype="random_string" length="30">
  <var name="rndclan" />
</setdynvars>

<request subst="true">
  <http url="/games/EGame/clans/%%_rndclan%" contents="clanName=%%_rndclan%&leaderName=Stoffe" content_type="application/x-www-form-urlencoded" method="PUT" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%%_csrf_token%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

<thinktime value="5" random="true"></thinktime>

<request subst="true">
  <http url="/users/iSpartan/clans" contents="clanName=%%_rndclan%&leaderName=Stoffe" content_type="application/x-www-form-urlencoded" method="POST" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%%_csrf_token%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

<thinktime value="5" random="true"></thinktime>

<request subst="true">
  <http url="/users/pewdiePie/clans" contents="clanName=%%_rndclan%&leaderName=Stoffe" content_type="application/x-www-form-urlencoded" method="POST" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%%_csrf_token%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

<thinktime value="5" random="true"></thinktime>

<request subst="true">
  <http url="/games/EGame/clans/%%rndclan%" method="PUT" version="1.1"></http>
</request>

<thinktime value="5" random="true"></thinktime>

<request subst="true">
  <http url="/users/iSpartan/clans/%%_rndclan%" method="PUT" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%%_csrf_token%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

<thinktime value="5" random="true"></thinktime>

<request subst="true">
  <http url="/users/pewdiePie/clans/%%_rndclan%" method="PUT" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%%_csrf_token%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

<thinktime value="5" random="true"></thinktime>

<request subst="true">
  <http url="/games/EGame/clans/%%_rndclan%" method="DELETE" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%%_csrf_token%" />
  </http>
</request>
<http_header name="Authorization" value="testToken"/>
</http>
</request>
</session>

<session name="http–GET–test" weight="8" type="ts_http">
  <request subst="true"> <http url="/users/iSpartan" method="GET" version="1.1">...
  </request>
  <thinktime value="2" random="true"></thinktime>
  <request subst="true"> <http url="/store.games/EGame/items" method="GET" version="1.1">...
  </request>
  <thinktime value="2" random="true"></thinktime>
  <request subst="true"> <http url="/store.games/EGame/clans" method="GET" version="1.1">...
  </request>
</session>

<session name="http–physical–order–test" weight="2" type="ts_http">
  <request subst="true">
    <match do="dump" when="nomatch">WIUDB</match>
    <dyn_variable name="csrf_token" jsonpath="token" />
    <http url="/getToken" method="GET" version="1.1"></http>
  </request>
</session>

<session name="http–friend–accept–remove" weight="5" type="ts_http">
  <request subst="true">
    <match do="dump" when="nomatch">WIUDB</match>
    <dyn_variable name="csrf_token" jsonpath="token" />
    <http url="/users/iSpartan/friends" method="GET" version="1.1"></http>
  </request>
  <thinktime value="2" random="true"></thinktime>
  <if var="friend_list neq="">
    <transaction name="remove_friends">
      <foreach name="friend" in="friend_list">
        <request subst="true">
          http url="/users/iSpartan/friends/%friend%" method="DELETE" version="1.1">
            <http_header name="X-CSRF-TOKEN" value="%%csrf_token%%" />
            <http_header name="Authorization" value="testToken" />
          </http>
        </request>
      </foreach>
    </transaction>
  </if>
</session>
<i f >

<!-- iterate over list and get each item -->
< i f var="pending_requests" neq=" ">
  <transaction name="accept_each_request">
    <foreach name="request" in=" pending_requests">
      <request subst="true">
        <http url="/users/iSpartan/friends/%%_request%%" method="PUT" version="1.1">
          <http_header name="X-CSRF-TOKEN" value="%_%csrf_token%_%" />
          <http_header name="Authorization" value="testToken" />
        </http>
      </request>
    </foreach>
  </transaction>
</i f >
</session>

<session name="http-friend-send-request" weight="5" type="ts_http">
  <request subst="true">
    <match do="dump" when="nomatch">WIUDB</match>
    <dyn_variable name="csrf_token" jsonpath="token" />  
    <http url="/getToken" method="GET" version="1.1"></http>
  </request>
</session>

<request subst="true">
  <http url="/users/iSpartan/friends" contents="myUsername=Stoffe" content_type="application/x-www-form-urlencoded" method="POST" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%_%csrf_token%_%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

<thinktime value="3" random="true"></thinktime>

<request subst="true">
  <http url="/users/iSpartan/friends" contents="myUsername=pewdiePie" content_type="application/x-www-form-urlencoded" method="POST" version="1.1">
    <http_header name="X-CSRF-TOKEN" value="%_%csrf_token%_%" />
    <http_header name="Authorization" value="testToken" />
  </http>
</request>

</session>
</sessions>

Listing B.1: The different test sessions used during testing
Appendix C
Test Cases

Listing C.1: OpenShift test case

Listing C.2: Local test case two

Listing C.3: Local test case one
Listing C.4: Local test case three

```xml
<servers>
  <server host="localhost" port="5000" type="tcp"/>
</servers>

<arrivalphase phase="1" duration="120" unit="second">
  <users maxnumber="1200" arrivalsequence="10" unit="second"/>
</arrivalphase>

<arrivalphase phase="2" duration="120" unit="second">
  <users maxnumber="1320" arrivalsequence="11" unit="second"/>
</arrivalphase>

<arrivalphase phase="3" duration="120" unit="second">
  <users maxnumber="1440" arrivalsequence="12" unit="second"/>
</arrivalphase>

<arrivalphase phase="4" duration="120" unit="second">
  <users maxnumber="1200" arrivalsequence="10" unit="second"/>
</arrivalphase>

<arrivalphase phase="5" duration="120" unit="second">
  <users maxnumber="1320" arrivalsequence="11" unit="second"/>
</arrivalphase>

<arrivalphase phase="6" duration="120" unit="second">
  <users maxnumber="1440" arrivalsequence="12" unit="second"/>
</arrivalphase>
</servers>
```

Listing C.5: Local test case four

```xml
<servers>
  <server host="localhost" port="5000" type="tcp"/>
</servers>

<arrivalphase phase="1" duration="240" unit="second">
  <users maxnumber="2400" arrivalsequence="10" unit="second"/>
</arrivalphase>

<arrivalphase phase="2" duration="240" unit="second">
  <users maxnumber="2640" arrivalsequence="11" unit="second"/>
</arrivalphase>

<arrivalphase phase="3" duration="480" unit="second">
  <users maxnumber="1440" arrivalsequence="12" unit="second"/>
</arrivalphase>

<arrivalphase phase="4" duration="240" unit="second">
  <users maxnumber="2400" arrivalsequence="10" unit="second"/>
</arrivalphase>

<arrivalphase phase="5" duration="240" unit="second">
  <users maxnumber="2640" arrivalsequence="11" unit="second"/>
</arrivalphase>

<arrivalphase phase="6" duration="240" unit="second">
  <users maxnumber="1440" arrivalsequence="12" unit="second"/>
</arrivalphase>
</servers>
```
Bibliography


[38] Nicolas Niclausse. Tsung is an open-source multi-protocol distributed load testing tool. 2013. URL: https://tsung.erlang-projects.org/ (visited on 02/19/2016).


Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/