Assignment 3

Software Architekturen
SS2010

Group: 49

- Patrick Marschik, Mat. Nr.: 0625039, Stud. Kz.: 066 933
- Martin Schwengerer, Mat. Nr.: 0625209, Stud. Kz.: 066 937
- Michael Vögler, Mat. Nr.: 0625617, Stud. Kz.: 066 937
- Matthias Rauch, Mat. Nr.: 0626140, Stud. Kz.: 033 534
- Benjamin Bachhuber, Mat. Nr.: 1028430, Stud. Kz.: 066 933

Contents

1 Technical Specification

Servers	
Application Server	Tomcat
DBMS	PostgreSQL 8.4
Cache	EhCache

Table 1: Server technology

Frameworks	
Dependency Injection	Spring
OR-Mapping	Hibernate
Cache	EhCache
MVC-Framework	Spring MVC & J-Query
Messaging	Spring Integration

Table 2: Frameworks

Tools	
Build System	Maven
IDE	IntelliJ
Version Control System	Git

Table 3: Tools

2 Prototype Architecture

2.1 (Entity) Class Diagram

The entities of the system are distributed over three different persistence units.

Map domain (see Figure 1) The map domain describes all the classes that are needed to play on a specific map.

Message domain (see Figure 3) The message domain consists of all classes that are persisted due to messaging.

User domain (see Figure 2) The user domain consists of all classes that are persisted by the user management component.

2.2 Component Diagram

2.3 Deployment Diagram

2.3.1 Uptime Calculation

The deployment diagram shows three main sites where artifacts have to be deployed.

1. There exists a cluster hosting Map Controller and the corresponding database for each map in the game. The cluster itself can be divided into several nodes that can server requests for this map. A load balancer is responsible for the request routing. The databases of the nodes must be replicated since they hold the same set of data.
2. A similar cluster hosting the User Management component serves registration and session management requests.
3. Another cluster is responsible for processing messaging requests. As denoted in D-1 there exists a message queue for each node of the cluster to provide reliable messaging. A redundant network of SMTP servers ensures availability of e-mail notifications.

This redundancy of services is necessary on the one hand to fulfill the performance requirements and provide a scalable architecture. On the other hand it results in a fail safe system as the following calculation shows:

- $P(nodeFails) = 0.1$

As the requirements specification denotes we can assume that each node has a failure probability of 10%.

A cluster consists of n nodes. Since the cluster is available when at least one node is available we can set failure possibility of a cluster.

- $P(clusterFails) = 0.1^n$

When we consider that we have three cluster in our system, we get the formula for the overall system fail rate:

- $P(clusterFails) = 3 * 0.1^n$

To calculate availability we have to consider the inverse probability:

- $P(systemAvailable) = 1 - 3 * 0.1^n$

If we solve the inequality

- $1 - 3 * 0.1^n \geq 0.99999$

we receive

- $n \geq 4.47712 \implies n \geq 5$

That means if that we reach an availability of $99,99\%$ if we have at least 5 nodes in each cluster. To fulfill security requirements the communication between server and client is always encrypted.

2.4 Architectural Decisions

2.4.1 D-1

Issue	There can be lots of simultaneous messages and not all of them can be handled by the database and mail servers directly.
Decision	Use an asynchronous message queue as buffer for sent messages. The middleware has to offer a queue for each Notification node.
Group	Component Interaction
Assumptions	• Lots of simultaneous messages • Not all of them can be handled directly by processing nodes
Constraints	-
Positions	• Directly send/store messages at the mail server resp. database server using explicit invocation. • Use one single message queue for all e-mails and use one single for all internal messages.
Argument	Message queues buffer messages to ensure the system can cope with load peeks. Since processing of the message is delayed a call to the Notification Component would take a much smaller amount of time. I also decided to use a queue for each processing node, since a central queue would cause a single-point of failure.
Implications	The middleware must be chosen appropriately to support asynchronous message queues.
Related decisions	-
Related requirements	• There can be lots of simultaneous messages and not all of them can be handled by the database and mail servers directly. • Make sure that notifications are reliable and do not simply rely on the database or, even worse, the mail server.
Related artifacts	requirements specification, component diagram, deployment diagram
Related principles	-

Table 4: Design decision - D-1

2.4.2 D-2

Issue	A complex system should be divided into components to enforce separation of concerns and provide reusability and modifiability.
Decision	Structure the architecture into layers, s.t. higher layers depend on lower layers.
Group	Component Interaction
Assumptions	• The functionality can be grouped into components. • The components can define interfaces to make their functionality externally available.
Constraints	-
Positions	• Use strong coupling between components. • Use a monolithic design.
Argument	Some low-level parts of the system (e.g. Persistence, Access Control) are used by many higher-level parts. Strong coupling between components would restrain us concerning modifications be done in future, since we could not exchange components. A monolithic design on the other hand restrains concerning distributability of the components.
Implications	The architecture should be grouped into the following layers (from high to low): 1. Presentation 2. Business Logic (Maps, Statistics, User Management) 3. Access Control 4. Cache 5. Persistence
Related decisions	-
Related requirements	-
Related artifacts	Component diagram
Related principles	Dependency Injection

Table 5: Design decision - D-2

2.4.3 D-3

Issue	Important actions have to be logged. It is also necessary to monitor system performance.
Decision	Use the Interceptor pattern.
Group	Adaption
Assumptions	There exist well-defined points where interceptors can be plugged in.
Constraints	-
Positions	• Use hard-coded logging in each component. • Use profiling tools to monitor application and database.
Argument	The interceptor pattern provides hooks, where additional functionality can be injected. Thus not only auditing is supported. In contrast to hard-coded logging, interceptors can also be injected at run-time, if the configuration supports it. Compared to profiling tools, interceptors support all execution environments and databases. Besides that profiling tools consume more resources.
Implications	Interceptors should be pluggable in the following scenarios: • user login/logout • action start/end • database access • notification sending/receiving • complex calculations (map generation, attacks, etc.)
Related decisions	-
Related requirements	Every important action in the system has to be logged. There should be a user ranking which can be seen by every user: user with most points, richest user, strongest troop type, and so on. Also try to monitor some aspects of the system performance (e.g., average processing times, resource usage) and the system configuration itself (e.g., currently active nodes). Try to keep this information as up-to-date as possible, but do not create it directly from live data.
Related artifacts	Requirements specification
Related principles	-

Table 6: Design decision - D-3

2.4.4 D-4

Issue	No other user should be able to access or manipulate sensitive data of other users.
Decision	Provide special interceptors to ensure access control.
Group	Adaptation
Assumptions	The system supports interceptors.
Constraints	-
Positions	• Implement access control in database. • Implement access control directly in the business logic.
Argument	Realizing access control via interceptors has two advantages compared with the other positions: Firstly as denoted in D-3 interceptors can be plugged into the system at run-time. This means changes in access control don't demand the rollout of a new version and therefore do not result in downtime. Secondly the business code isn't messed up with security specific code and therefore easier to read, which directly corresponds to better maintainability and a lower bug rate.
Implications	Security interceptors should be placed at the following scenarios: • action start • database access
Related decisions	D-3
Related requirements	The system has strong security requirements, and you should prevent users from cheating or manipulating the game. No other user should be able to access or manipulate sensitive data of other users.
Related artifacts	Requirements specification
Related principles	-

Table 7: Design decision - D-4

2.4.5 D-5

Issue	Performance is crucial. The system needs to handle 1000's of concurrent users.
Decision	It should be possible to partition the system horizontally for each map. That means that each map should have a denoted server (or server farm). Besides that each node has a cache to minimize database roundtrips.
Group	Performance
Assumptions	The system can be partitioned horizontally for each map.
Constraints	-
Positions	• Set up a single server that is strong enough to handle 1000's of users. • Set up many nodes that mirror the whole database.
Argument	Distributing the application logic over more than one server is a good idea for scenarios where performance and availability are key requirements. Concerning availability clustering has the benefit that we don't have a single point of failure. Besides that in our concrete scenario, the map is the perfect choice for a partition criteria, since there are no cross-map operations possible by requirements specification. This results in less replication overhead.
Implications	Load balancing
Related decisions	D-2 (cache layer)
Related requirements	Performance must be consistent. It is not acceptable for a user to have to wait more than two or three seconds when submitting a post or loading a page. So think about a good strategy how to scale all parts of the application.
Related artifacts	Requirements specification, deployment diagram
Related principles	-

Table 8: Design decision - D-5

2.4.6 D-1

Issue	TODO
Decision	TODO
Group	TODO
Assumptions	TODO
Constraints	TODO
Positions	TODO
Argument	TODO
Implications	TODO
Related decisions	TODO
Related requirements	TODO
Related artifacts	TODO
Related principles	TODO

Table 9: Design decision - D-1

2.5 Future work

- Load-Balancer?
- https?

3 Prototype Installation Guidelines

3.1 Requirements

- Tomcat 6 or 7 running on `localhost:8080/`
- PostgreSQL 8.4 running on `localhost:5432/`
  - Database: `swa`
    - accessible by user: `swa`
    - password for user: `swa11`
- Ant or Maven

Run quartz script!! Run log4j script!!

3.2 Compiling and Deploying

asdf

3.3 Entry Point

The entry point to play **SWAG** is http:localhost:8080/user/swag/user/. There you can register an user, login and choose a map to play on.

A Figures

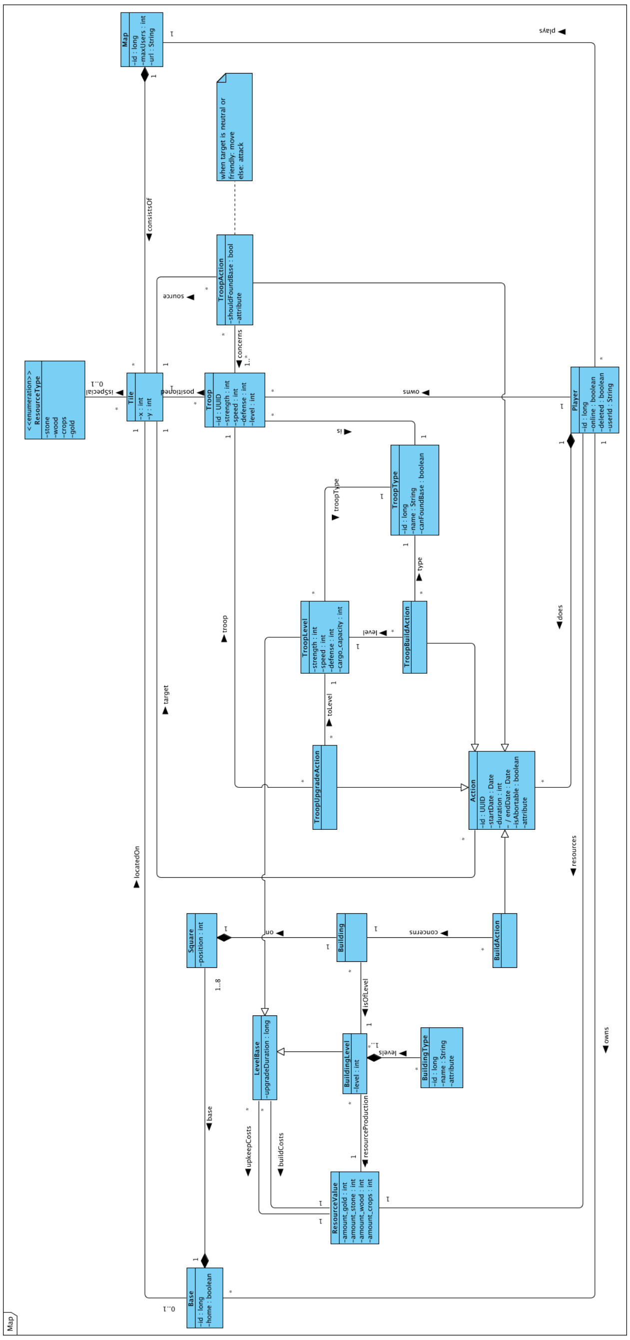

Figure 1: SWAG - Map Class Diagram

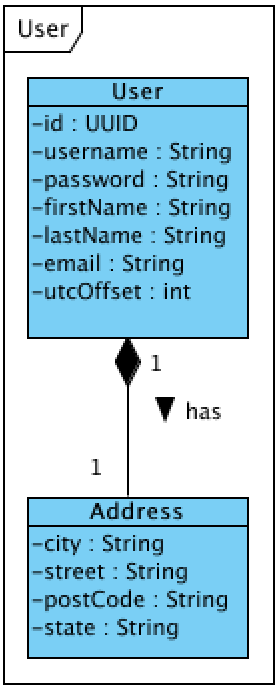

Figure 2: SWAG - User Class Diagram

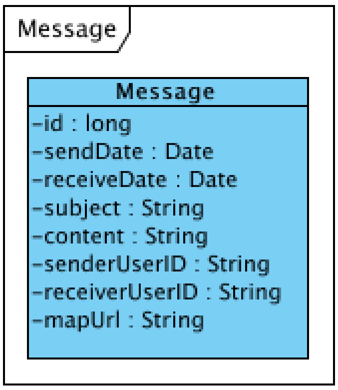

Figure 3: SWAG - Message Class Diagram