

# Kurzbericht

## SnakeCore, SnakeFX, SnakeServer

Ostfalia Fachhochschule für angewandte Wissenschaften

Benjamin Wulfert  
Leonard Reidel

Semester: Wintersemester 2020

20. Januar 2021

## Inhaltsverzeichnis

Einleitung.....	4
Modul-Architektur.....	4
SnakeCore.....	4
SnakeFX - Frontend.....	4
SnakeServer - Backend.....	5
SnakeTest – Test-Modul.....	5
SnakeFX - Front End.....	6
Architektur.....	6
BaseController.....	6
BaseApplication.....	7
Communication / Information-Transport.....	7
User-Interface.....	9
Login-Screen.....	9
Home-Screen.....	9
Spielhistorie-Screen.....	9
New-Game-Screen.....	9
Snake-Implementierung.....	10
Schlangenexistenz und -bewegung.....	10
MapEntity - Power-Ups & Food-Element Erzeugung.....	11
Aktionen (Schlange, trifft Wand, andere Schlange, Essen, sich selbst.....)	12
SnakeServer.....	14
Sourcecode-Packages.....	14
Persistenz-Layer.....	14
API-Layer - HTTP-Schnittstelle / RESTful Webservice.....	15
HTTP-Endpoints.....	15
STOMP-Endpoints.....	16
Communication / Information-Transport.....	16
Umsetzung der Anforderungen.....	19
Accountverwaltung.....	19
Spielhistorie.....	19
Erstellung und Wiedergabe von Sound-Effekten.....	19
Entwurfsmuster.....	20
Factory-Pattern.....	20
Observer-Pattern.....	20

Composite-Pattern.....	20
Erweiterungen und Extras.....	21
Spiel- und Spieler-Management / Lobby-Metapher.....	21
Chat-System / Diagnostik zwischen Client und Server.....	21
Nicht realisierte Anforderungen.....	21
Bedienungsanleitung und Spielregeln.....	22
Einrichtung.....	22
Anmeldung / Registrierung.....	22
Homescreen – Lobby, Chat-System, Spiel-Management, Spielhistorie.....	23
Game-Screen.....	26
Projektplan.....	27
Resümee & Ausblick.....	29
Verwendete Software.....	29

## Einleitung

Dieses Dokument stellt den Kurzbericht für das Projekt des Modul Patterns und Frameworks dar. Im Folgenden werden die verschiedenen Aspekte des Projekts beschrieben. Des Weiteren wird in diesem Kurzbericht dargelegt wie die Anforderungen des Projekts umgesetzt wurden sowie die Erweiterungen vorgestellt, Anleitung zur Bedienung gegeben und ein aktualisierter Projektplan dargestellt.

Das gesamte Projekt ist in der Programmiersprache Java entwickelt worden. Alle Module des Projekts werden mit Java 8 - und Oracles Java Development Kit kompiliert, getestet und ausgeführt. Als Build-Management-System des Projekts kommt Apache Maven 3 zum Einsatz. Jedes Modul enthält daher eine pom.xml welche das jeweilige Project-Object-Model (Attribute der Module sowie deren Software-Abhängigkeiten) beschreibt.

Der Quelltext des Projekts sowie alle während der Umsetzung angefallenen Dokumente (Text-Dokumente, UML-Diagramme, Gantt-Diagramme, etc.) können den folgenden GitHub-Repositories entnommen werden:

<https://github.com/benjaminfoo/SnakeFX> - Benjamin Wulfert

[https://github.com/Bummelnderboris/Patterns\\_and\\_Frameworks](https://github.com/Bummelnderboris/Patterns_and_Frameworks) - Leonard Reidel

## Modul-Architektur

Das folgende Schaubild stellt die Modul-Architektur des Systems dar.

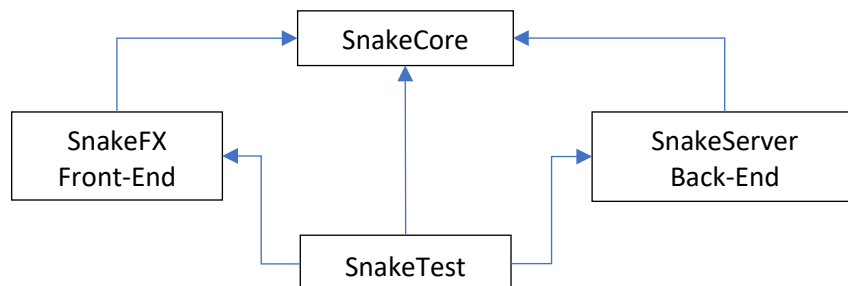


Abbildung 1 - Modul-Architektur des Anwendungs-Systems

## SnakeCore

Das Core-Modul enthält die Kern-Aspekte der Anwendung – dazu zählen beispielsweise Ausschnitte der Modelle welche im Klassendiagramm enthalten sind, die gemeinsam genutzten Endpoints der Schnittstelle sowie Konstanten welche sowohl im Backend als auch im Frontend verwendet werden. Das Core-Modul enthält also Programmteile welche sowohl im Front- als auch im Backend verwendet werden. Des Weiteren enthält das Core-Modul die verschiedenen Implementierungen von Food- und Power-Up MapEntities. Ein MapEntity stellt die Elternklasse für die Klassen FoodEntity, PowerUp-Entity, FreezeOtherPlayer-Entity oder Predator-Entity dar – die konkreten Entities werden im Laufe der Ausarbeitung ausführlich beschrieben.

## SnakeFX - Frontend

Das Modul SnakeFX ist das Front-End der Anwendung. Im Front-End sind die User Interfaces (UI) definiert und implementiert. Des Weiteren konsumiert das Front-End mittels REST-Schnittstelle

Daten aus dem Backend. Die Implementierung und die gesamten Mechaniken des Snake-Spiels sind ebenfalls Teil des Front-Ends.

### SnakeServer - Backend

Das Modul SnakeServer enthält alle Aspekte des Backend – dazu zählen die Persistenz-Schicht der Anwendung welche mit Spring Data JPA / Hibernate realisiert werden, so wie eine in Java implementierte Datenbank welche direkt mit dem Backend initialisiert wird (H2) – die Vorteile dieses Vorgehens werden weiteren Verlauf des Dokuments dargestellt. Ein weiterer Aspekt des Backend ist die Bereitstellung der REST-Schnittstelle sowie die Auslieferung der Persistenz-Daten darüber. Der letzte Aspekt des Backend ist die direkte bidirektionale Kommunikation mit den angemeldeten Clients über Web-Sockets um Eingaben der Spieler entgegenzunehmen und an die Teilnehmer einer Runde zu replizieren.

### SnakeTest – Test-Modul

Das Modul SnakeTest enthält Implementierungen für verschiedene Test-Szenarien und besitzt aus diesem Grund Referenzen auf alle weiteren Untermodule. Dies geschieht auf der Notwendigkeit sowohl alle einzelnen Aspekte pro Modul als auch das Zusammenspiel des gesamten Systems zu testen. Das Modul enthält unter anderem verschiedene Test-Implementierungen welche automatisch eine Backend-Instanz und eine bis mehrere Front-End-Instanzen (initialisieren, des Weiteren werden folgende Schritte automatisiert ausgeführt (im Folgenden werden die zwei Front-End-Instanzen mit F1 und F2 bezeichnet):

1. F1 und F2 - Registrierung eines neuen Accounts
2. F1 und F2 - Login am Backend mit zuvor erstellten Account
3. F1 – Erzeugen eines Spiels in der Lobby, Teilnahme
4. F2 – Teilnahme an erzeugten Spiel
5. F1 – Start des Lobby-Spiels
6. F1 und F2 – Start des Snake-Spiels
7. F1 und F2 – Solange kein Gewinner existiert: Führe Eingaben durch und synchronisiere diese Eingaben via Backend

## SnakeFX - Front End

Die folgenden Abschnitte beschreiben die Architektur sowie die realisierten Funktionalitäten und Prozesse des Front-Ends.

### Architektur

Die folgenden Abschnitte beschreiben die Software-Architektur des Front-Ends.

#### BaseController

Das Frontend besitzt eine auf Vererbung basierende Controller-Hierarchie. Jeder im Front-End verwendete Controller erbt von BaseController - einer abstrakten Oberklasse welche verschiedene Methoden besitzt und mittels Vererbung jedem ableitenden Controller zur Verfügung stellt. Dazu zählen Beispielsweise:

- das Laden von Szenen (Scenes) mittels FXML-Datei (JavaFX Markup Language) in derselben Stage / Window.
- verschiedene UI-bezogene Mechanismen wie das Dekorieren der Stage mittels Icon oder dem Aktualisieren des Stage-Titles, etc.
- Eine Referenz auf die (JavaFX-)Application selbst welche im Folgenden beschrieben wird.
- Life-Cycle Methoden (Initialize und PostInitialize) zur Definition von Callbacks pro JavaFX-Stage

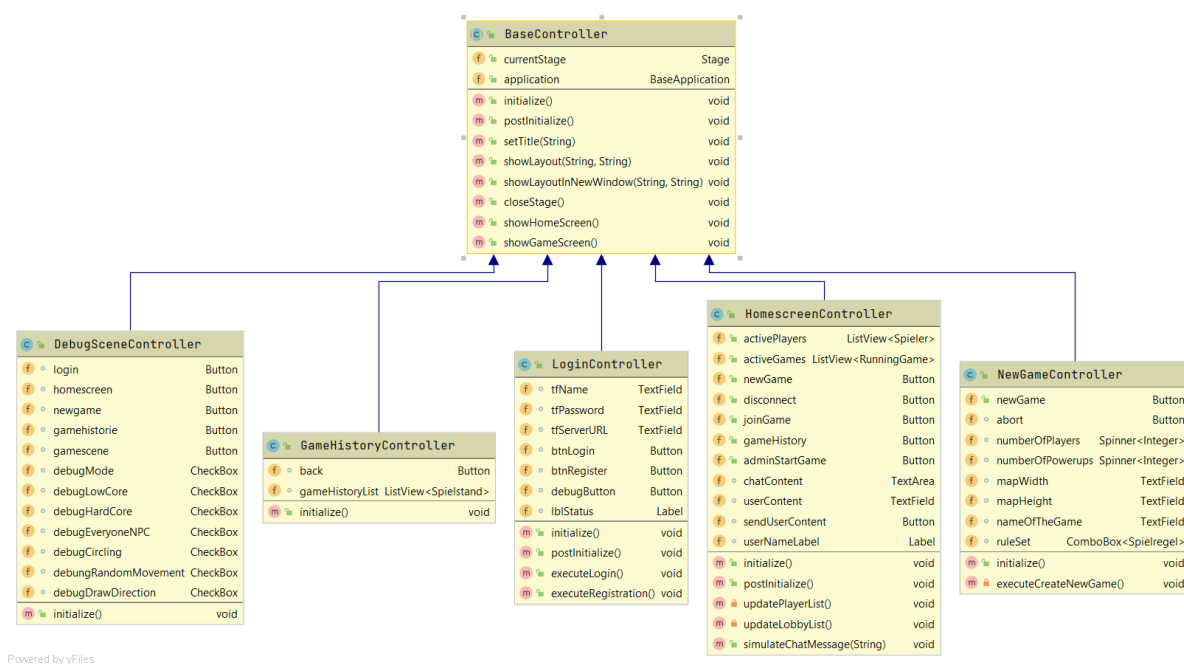


Abbildung 2 - Klassendiagramm der implementierten Controller-Architektur des Front-Ends

## BaseApplication

JavaFX-Anwendung besitzen als kleinsten gemeinsamen Nenner eine Vererbung zur Application-Klasse - diese Klasse selbst kann jedoch auch eine abgeleitete Klasse von Application sein. Dieser Sachverhalt kann genutzt um einen einzelnen Bezugspunkt logisch zusammenhängender Mechanismen zu bilden.

Aus diesem Grund wird im SnakeFX-Projekt die Ausgangsklasse von BaseApplication abgeleitet. Diese Klasse besitzt, ähnlich wie die BaseController, verschiedene Mechanismen welche jedoch unabhängig vom User-Interface verwendet werden können. Darunter fallen Beispielsweise:

- Eine Referenz auf den StompClient welcher genutzt werden kann um mittels WebSockets mit dem Backend zu kommunizieren
- Der UserConfig welche die Anmelde- und Session-Daten des aktuellen Users besitzt
- Einen Controller-Cache welcher implizit genutzt wird um Controller-Instanzen wiederzuverwenden

Mithilfe dieser Vererbungshierarchie ist es auf einfache Art und Weise möglich mittels Betätigung einer Schaltfläche (bspw. der Betätigung eines Buttons) einen HTTP- oder eine WebSocket-Message an das Backend zu initiieren, Operationen im selben Fenster auszuführen, etc. - und da jeder Controller von BaseController erbt und dieser eine Referenz auf die BaseApplication besitzt können auch verschiedene Aspekte des Front-Ends gesteuert und kontrolliert werden.

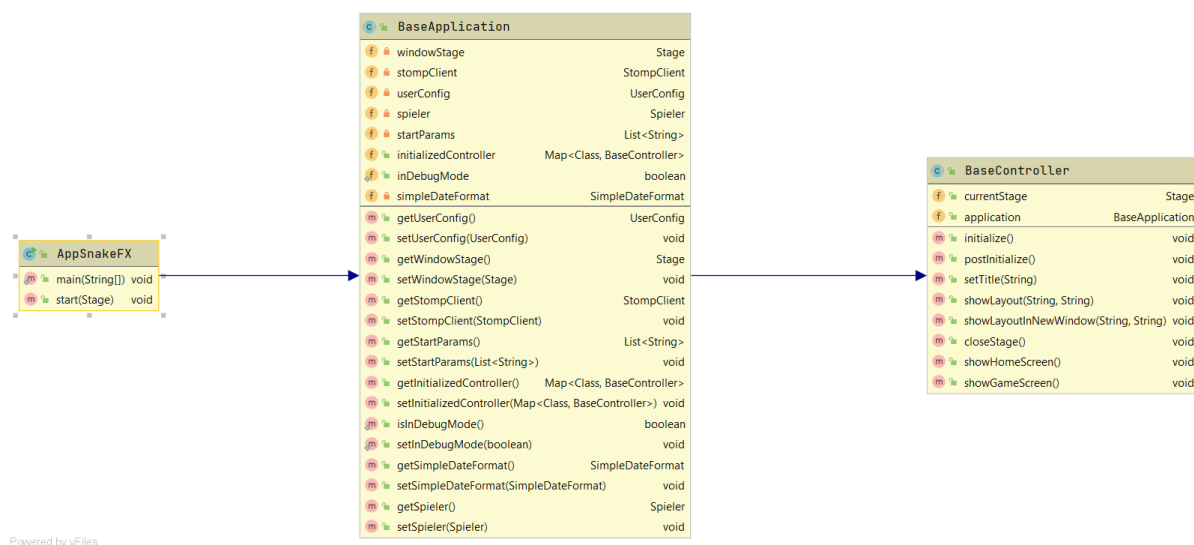


Abbildung 3 - Klassendiagramm welches die Relation zwischen Anwendung und BaseController darstellt.

## Communication / Information-Transport

Das Front-End ist sowohl in der Lage eine synchrone Kommunikation mittels HTTP als auch eine asynchrone Kommunikation mittels STOMP/WebSockets durchzuführen.

Die synchrone Kommunikation über das HTTP wird mit der Software-Bibliothek **Unirest** realisiert. Dieser HTTP-Client stellt eine Implementierung verschiedener HTTP-Mechanismen bereit – so kann dieser HTTP-Requests an eine vorgegebene URL senden und die darauf folgende Antwort, die HTTP-Response empfangen und verarbeiten. Im Kontext des Anwendungs-Systems wird Unirest für den

Transport von Registrierungs- und Login-Daten verwendet. Des Weiteren wird der initiale Bezug von Lobby- und Spielhistorie-Daten verwendet (synchron).

Die asynchrone Kommunikation erfolgt mittels STOMP/WebSockets – diese Funktionalität wird mithilfe der Software-Bibliothek **Spring-Messaging** realisiert. Wie bereits beschrieben werden die Lobby- und Spielhistorie-Daten initial über das HTTP bezogen – alle zur Laufzeit auftretenden Daten-Änderungen (bspw. der Lobby- oder Spielhistorie-Daten ) werden mittels WebSockets in Echtzeit kommuniziert und im Client entsprechend dargestellt. Des Weiteren erfolgt die Kommunikation der Spieldaten (Eingabe der Bewegungsdaten von Spielern, Position des Foods und der Power-Ups, etc.) ebenfalls asynchron über WebSockets.

Die folgenden Befehle werden für den Datenaustausch verwendet und können von implementierenden Clients aufgerufen werden:

**connect** - Stellt eine Verbindung mittels StompClient auf einen HTTP-Handshake Endpoint her. Die dabei aufgebaute Session wird für den im folgenden beschriebenen Informationsaustausch verwendet.

**subscribe** - Führt eine Subscription (dt. etwa "Anmeldung") auf ein bestimmtes **Topic** (dt. etwa "Thema") aus - der StompClient bspw. subscribed auf den "/topic/players"-Endpoint - im Anschluss werden alle Änderungen des Topics via Publish-Mechanismus an den Client übertragen und dort mittels Callbacks genutzt um das User-Interface zu aktualisieren.

**unsubscribe** - Führt eine Unsubscription (dt. etwa "Abmeldung") auf ein bestimmtes Topic aus.

**send** - Sendet eine Nachricht in einem definierbaren Message-Format an das Backend - die Nachrichten-Inhalte werden mittels Jackson im JSON-Format serialisiert und erzeugt, und auf der jeweiligen Gegenseite (Client <-> Backend) de-serialisiert (Un/marshalling).

Wie in den Klassendiagrammen dargestellt besitzt die BaseApplication eine Instanz des StompClients welche für die zuvor vorgestellten Mechanismen zur Kommunikation mit dem Backend verwendet werden kann.

Jede synchrone HTTP-Anfrage wird im Front-End innerhalb eines eigenständigen Threads ausgeführt. Durch die Verwendung dieser Art von **nebenläufiger Programmierung** wird das User-Interface der Anwendung nicht blockiert was dem Benutzer den Vorteil bietet, dass die Anwendung trotz synchroner Anfragen weiterverwendet werden kann ohne explizit auf Antworten des Backends warten zu müssen. Dieses Merkmal stellt eine Eigenschaft responsiver Benutzeroberflächen dar und gilt in der Entwicklung von User-Interfaces Aufgrund des positiven Nutzererlebnisses (UX = User Experience) als erstrebenswert.



## User-Interface

Der folgende Abschnitt beschreibt die Benutzeroberfläche des Front-Ends. Die Benutzeroberfläche der Anwendung ist in verschiedene Scenes (Szenen) unterteilt. Jede Schaltfläche (Buttons, Textfelder, ListViews, etc.) wird als Actor (Schauspieler) bezeichnet, sodass jeder Actor pro Scene seine Tätigkeit (Darstellung von Informationen, Reaktion auf bestimmte Events) durchführt. Jede Scene wird wiederum in einer gemeinsamen Stage (Bühne) dargestellt – die Begrifflichkeiten stellen eine Analogie aus dem Bereich des Schauspiels und dem Theater dar – in dem verschiedene Schauspieler zusammen in einer Szene wirken und mehrere Szenen auf einer gemeinsamen Bühne dargestellt werden. Im Front-End werden alle Scenes als Screen bezeichnet, da jeder Screen eine eigene Scene darstellt welche jeweils auf derselben Stage durchgeführt werden.

### Login-Screen

Der Login-Screen enthält vier Actors / Schaltflächen für die Interaktion des Benutzers. Zwei Texteingaben zur Angabe eines Benutzernamen und ein Passwort – und zwei Schaltflächen / Buttons um die Benutzerangaben (Name, Passwort) mittels Login oder Registrierung an das Backend zu übertragen. Betätigt der Benutzer die Registrierungs-Schaltfläche werden die vergebenen Benutzerdaten an das Backend übertragen und in der Datenbank persistiert. Betätigt der Benutzer die Login-Schaltfläche so werden die Benutzerdaten für einen Datenabgleich innerhalb der Datenbank durchgeführt – existiert ein Benutzer mit jeweiligen Benutzerdaten gilt der Login als erfolgreich, worauf das Front-End zum Home-Screen wechselt.

### Home-Screen

Der Home-Screen stellt die zentrale Benutzeroberfläche der Anwendung dar und bietet die Möglichkeit, alle angemeldeten Spieler und alle zu startenden oder bereits laufenden Spiele zu betrachten. Des Weiteren gelangt der Benutzer über den Home-Screen zur Spielhistorie-Oberfläche. Durch Betätigung der Schaltfläche „Neues Spiel“ ist der Benutzer in der Lage neue Spielrunden zu definieren und in der Lobby zu veröffentlichen. Aktive Spieler können dann, sofern noch genügend Kapazitäten vorhanden sind, dem Spiel beitreten (durch Betätigung der „Teilnehmen“ Schaltfläche rechts im Menü). Des Weiteren ermöglicht die „Abmelden“-Schaltfläche die Abmeldung eines Benutzer vom System inklusive verschiedener De-Initialisierungs-Aufgaben wie der Abmeldung von Topics, etc. – dieses Thema wird in den folgenden Kapiteln näher beschrieben.

### Spielhistorie-Screen

Die Benutzeroberfläche für die Spielhistorie zeigt dem Benutzer alle in der Vergangenheit gespielten Spiele an welche im Backend persistiert wurden. Die Daten werden über das HTTP vom Backend bezogen und in einer ListView dargestellt – jedes persistierte Spielergebnis stellt dabei ein Element in der ListView dar. Sowohl im Spielhistorie-Screen als auch im Homescreen werden eigene Cell-Renderer Implementierungen verwendet um die verschiedenen Informationen gezielt darstellen zu können.

### New-Game-Screen

Der New-Game-Screen erlaubt es einem Anwender ein neues Spiel zu definieren und diese Definition in der Lobby des Backend zu posten. Interessierte Spieler können an einem veröffentlichten Spiel teilnehmen. Der Anwender welcher das Spiel definiert hat gilt als Admin / GameMaster und kann

dadurch den Zeitpunkt des Spielstarts bestimmen. Innerhalb des New-Game-Screens können verschiedene Parameter konfiguriert werden wie bspw. die Anzahl maximaler Spieler, die Anzahl maximal auftretender Food-Elemente oder Power-Ups sowie die für die Spielrunde geltende Spielregel.

### Snake-Implementierung

Für die Visualisierung des Snake-Spiels wird ein Spielfeld anhand von Höhen- und Breiten-Parameter definiert und ein zwei dimensionales euklidisches Koordinaten-System aufgespannt. Dieses Spielfeld wird innerhalb einer Canvas (einem dediziert programmierbaren Zeichenbereich) dargestellt. Das Spielfeld wird dabei innerhalb einer Scene dargestellt, so wie alle anderen Screens des Front-Ends auch. Die im folgenden beschriebene Anwendungs-Logik wird in der Klasse GameController definiert und bei Wechsel in den Game-Screen durchgeführt.

Für die kontinuierliche Aktualisierung der Spiel-Logik und der Darstellungs-Logik wird eine Timeline verwendet welche in einem frei wählbaren Intervall einen kontinuierlichen Methoden-Aufruf durchführt. Im Kontext der Programmierung von Spielen wird so ein Vorgehen als Update-Cycle / Update-Loop bezeichnet. Mit dem Start der Timeline (mittels `new Timeline().start(t -> update())`) wird die Spiellogik gestartet – der Zustand des Spiels aktualisiert und letztlich innerhalb des Canvas-Actors visualisiert. Dabei wird jede Durchführung der Update-Methode als *Tick* bezeichnet.

### Schlangenexistenz und -bewegung

Für die Berechnung der Schlangen-Positionen und der Bewegungen wird wie zuvor erwähnt ein traditionelles zweidimensionales euklidisches Koordinatensystem verwendet. Die Elemente einer Schlange werden durch eine Liste von Vektoren mit jeweils zwei Komponenten dargestellt (x- und y-Komponente). Diese Komponenten werden mit einem zur Laufzeit berechneten Skalar skaliert um somit eine geeignete Größe für das Spielfeld darstellen zu können.

### Schlangeninitialisierung

Anhand einer x-y-Koordinate wird ein Startpunkt definiert. Des Weiteren werden für die Anzahl der initialen Schlangenlänge (`snake.initialLength`) weitere Schlangenkörper-Elemente instanziiert. Zu Beginn eines jeweiligen Spiels werden alle Schlangenkörper-Elemente mit einer vom Startpunkt abhängigen berechneten Position mit einer validen Position im Koordinaten-System versehen.

Des Weiteren wird eine Farbe für die Darstellung des Spielers übergeben womit jeder Spieler eindeutig identifiziert werden kann.

```
public Snake(Vector2 spawn, SnakeColor color) {
    for (int i = 0; i < initialLength; i++) {
        Vector2 bodyPart = new Vector2(x: -1, y: -1);
        body.add(bodyPart);
    }
    head = body.get(0);

    head.x = spawn.x;
    head.y = spawn.y;

    this.color = color;

    isPredator = false;
}
```

Abbildung 4 - Initialisierung einer Schlangen-Instanz

### Schlangenbewegung

Die Bewegung der Schlange wird berechnet, indem über jedes Listenelement des Schlangenkörpers iteriert wird: Bei jeder Iteration wird dem List-Element die x-y-Koordinate des in der Liste vorherigen Elements übertragen, sodass eine zusammenhängende für das Snake-Spiel typische Bewegung entsteht.

```
for (Snake snake : snakeList)
{
    // calculate the current position for each snake
    for (int i = snake.body.size() - 1; i >= 1; i--) {
        snake.body.get(i).x = snake.body.get(i - 1).x;
        snake.body.get(i).y = snake.body.get(i - 1).y;
    }
    snake.head = snake.head.add(snake.currentDirection);
}
```

Abbildung 5 - Übertragung der Schlangenkörper-Positionen an Vorgänger-Elemente innerhalb der Liste

Die Berechnung der jeweils neuen Positionen wird durchgeführt in dem ein Richtungsvektor auf den Kopf der Schlange addiert wird. Durch die Iteration über die Schlangenkörper-Elemente und die damit einhergehenden Übertragungen der Positions-Vektor entsteht der Mechanismus eine Schlange steuern zu können. Durch Betätigung der Pfeiltasten (oder alternativ W,A,S,D) wird der Richtungsvektor einer Schlange definiert – so verändert sich der Richtungsvektor einer Schlange durch Betätigung der Pfeilhoch-Taste (oder alternativ W) auf X = 0, Y = -1 (-1 da das Koordinatensystem an der Y-Achse gespiegelt wird) – durch Betätigung der Pfeiltaste-Links wird der Richtungsvektor beispielsweise auf X = 1, Y = 0 gewechselt.

Durch die Implementierung eines EventHandlers im GameController können die Tastatur-Eingaben eines Spielers erfasst und dementsprechend darauf reagiert werden – bei jeder Tastatur-Eingabe wird ein KeyEvent an den Listener übergeben, wobei jedes KeyEvent einen KeyCode enthält und jeder KeyCode für eine Taste auf der Tastatur steht. Den KeyCodes Pfeiltaste Hoch (KeyCode.UP), Pfeiltaste Runter (KeyCode.DOWN), Pfeiltaste Links (KeyCode.LEFT), Pfeiltaste Rechts (KeyCode.RIGHT) (oder alternativ W,A,S,D) wird wie zuvor beschrieben ein Richtungsvektor zugeordnet und dieser zur Berechnung der Schlangenbewegung verwendet.

### MapEntity - Power-Ups & Food-Element Erzeugung

Power-Ups sowie Food-Elemente werden über eine Liste / Set (HashSet) von MapEntity-Instanzen realisiert. Dabei enthält jede Instanz einer MapEntity eine javafx.scene.Image-Instanz welche das jeweils auf dem Spielfeld darzustellende Bild / Bitmap enthält, sowie einem Vektor um die Position der Darstellung zu bestimmen. Befindet sich ein Schlangenkopf auf derselben Koordinatenposition wie eine MapEntity-Instanz so wird ein Listener / Callback ausgeführt welches eine bestimmte Reaktion auslöst. Befindet sich der Schlangenkopf bspw. auf derselben Koordinate wie der einer MapEntity-Instanz vom Typ „Food“ wird die auslösenden Schlange um ein weiteres Körperteil ergänzt sowie der Score zur Spieler zugehörigen Schlange erhöht.

Die Arten und Positionen der MapEntities werden dabei vom Backend generiert sodass in jeder Instanz eines Front-Ends dieselben Daten zur Darstellung verwendet werden. Des Weiteren wird bei

jeder Berechnung berücksichtigt, dass nicht mehrere Elemente auf derselben Position erzeugt werden.

*Aktionen (Schlange, trifft Wand, andere Schlange, Essen, sich selbst...)*

Eine jede Aktion lässt sich dann - gleich der Essensaufnahme - über eine Überprüfung, ob ein Schlangenkopf sich auf bestimmten Koordinaten befindet, implementieren. Je nach Anwendungsfall können verschiedene Regeln in Kraft treten.

```
public void checkGameOver() {  
    // check for wall collision  
    for (Snake snake : snakeList) {  
        if (snake.head.x < 0 || snake.head.y < 0 ||  
            snake.head.x > config.rows - 1 ||  
            snake.head.y > config.columns - 1) {  
            gameOver = true;  
        }  
        // destroy itself  
        for (int i = 1; i < snake.body.size(); i++) {  
            if (snake.head.x == snake.body.get(i).getX() && snake.head.getY() == snake.body.get(i).getY()) {  
                gameOver = true;  
                break;  
            }  
        }  
    }  
}
```

Abbildung 6 - Beispiel: GameOver bei Kollision mit dem Rand des Spielfelds und einer Schlange oder einer Schlange mit sich selbst

```
// hitting a wall teleports a player to the other side  
if (snake.head.x < 0) {  
    snake.head.x = config.rows - 1;  
}  
if (snake.head.y < 0) {  
    snake.head.y = config.columns - 1;  
}  
if (snake.head.x > config.rows - 1) {  
    snake.head.x = 0;  
}  
if (snake.head.y > config.columns - 1) {  
    snake.head.y = 0;  
}
```

Abbildung 7 - Aktuelle Alternative - Befindet sich eine Schlange am Bildschirmrand so werden die Positionen der Schlange am anderen Ende des Spielfelds erscheinen

Beispiel: Schlange beißt Schlangenkörper-Element einer anderen ab

Wenn der Kopf einer Schlange den Körper einer anderen trifft, wird der Index des Eintrags der anderen Schlange ermittelt, dann die Gesamte Listenlänge Minus dem Index gerechnet und die daraus entstehende Zahl als Elemente der beißenden Schlange angehängt, indem die Koordinate der beißenden Schlange x mal in die Liste eingefügt wird (ähnlich der Initialisierungsmethodik). Da dies durch ein Power-Up realisiert werden kann, haben Schlangen das Attribut isPredator und durch den Konsum ein solches MapEntites vom Typ PowerUp auf den Wahrheitswert „true“ gesetzt werden kann. Entspricht der Wahrheitswert „true“ wird die jeweilige Logik aktiviert und wird mit dem Konsum eines anderen Power Ups wieder auf „false“ gesetzt.

Schlange konsumiert Körperelement einer anderen Schlange:

```
//check that predator snake doesn't eat itself
/*
for (Snake snake : snakeList) {
    for (Snake otherSnake : snakeList) {

        if (otherSnake != snake) {

            for (Vector2 part : otherSnake.body) {
                if (snake.head.equals(part)) {
                    if (!snake.isPredator) {
                        checkGameOver();
                    } else {
                        int totalLength = otherSnake.body.size();
                        int splittingPoint = otherSnake.body.indexOf(part);
                        int growth = totalLength - splittingPoint;

                        for (int i = splittingPoint; i < totalLength; i++) {
                            //cut bitten Snakes body
                            otherSnake.body.remove(i);

                            //add to biting Snake
                            Vector2 newPart = new Vector2(-1, -1);
                            snake.body.add(newPart);
                        }
                    }
                }
            }
        }
    }
}
```

Abbildung 8 - Implementierung der Predator-Logik und Veränderung der Körperelemente der jeweiligen Schlangen

## SnakeServer

Das Snake-Server Modul stellt das Backend der Anwendung dar. Teil des Moduls ist eine Datenbank sowie deren Anbindung an das Backend. Des Weiteren stellt SnakeServer die verschiedenen Schnittstellen zur Verfügung welche zur Kommunikation mit dem Front-End benötigt werden.

Das Backend setzt auf verschiedene Module des Spring-Ökosystems:

- Spring Data JPA - zum Aufbau und Management des Persistenz-Layers
- Spring Web - zur Bereitstellung von Web-Inhalten und REST-Services
- Spring Messaging - zur Bereitstellung und Kommunikation mittels WebSockets / STOMP

## Sourcecode-Packages

Wie in der Einleitung bereits erwähnt wurde das gesamte Anwendungs-System in der Programmiersprache Java entwickelt. In allen Modulen wird eine Package-Struktur verwendet welche die jeweiligen logischen Aspekte des Systems trennen – die folgenden Packages sind im Quelltext des Backends enthalten:

Das Package **controller** enthält verschiedene Controller (im Sinne der Model-View-Controller Architektur) welche bestimmte fachliche / technische Aspekte des Anwendungs-Systems realisieren. Dazu zählen Controller welche Anwendungslogik enthalten - Controller welche (mittels Spring Web) RESTful Webservices implementieren und somit die Kommunikation mittels HTTP ermöglichen - sowie Controller welche (mittels Spring Messaging) STOMP-Endpoints darstellen.

Im Package **persistence** existieren verschiedene Definitionen von Repository-Interfaces, welche, wie der Name vermuten lässt, für konkrete Implementierungen des Repository-Patterns genutzt werden - Spring Data generiert die konkreten Implementierungen anhand von Interface-Definitionen selbstständig wodurch der Implementierungsaufwand für ORM-bezogene Tätigkeiten stark reduziert werden kann. Mithilfe dieser Implementierung können CRUD-Aufgaben für verschiedene fachliche Modelle durchgeführt werden (bspw. speichern von Instanzen der Spielhistorie, Bezug von Spielhistorie-Daten und anschließender Bereitstellung im API-Layer / HTTP via RESTful-Webservices).

Im Package **runner** befinden sich verschiedene Ausprägungen von Spring-ApplicationRunnern welche bspw. genutzt werden um Initialisierungsaufgaben durchzuführen (z.B. Test-Daten in der Datenbank zu initialisieren).

Im Package **ws.server** befinden sich verschiedene Spring-Konfigurationsklassen welche zur Initialisierung von WebSockets und Security-Aspekten genutzt werden.

## Persistenz-Layer

Der Persistenz-Layer verwaltet die Speicherung, Aktualisierung und den Bezug von Daten aus dem relationalen Datenbank Management-System (RDBMS).

Als RDBMS wird die Database Engine H2 verwendet – welches vollständig in der Programmiersprache Java entwickelt wurde – diese bedeutet, dass die Datenbank-Engine als Teil der Anwendung definiert und gestartet werden kann. Dieses Vorgehen erweist sich insbesondere für das Aufsetzen des Workspaces und der Anwendung an sich als Hilfreich, da die zeitintensive Installation und

Konfiguration des RDBMS entfällt. Des Weiteren vereinfacht das Vorgehen die Realisierung und Nutzung von Unit-Tests.

Für die Realisierung des Persistence-Layers wird das Spring-Modul „Spring-Data JPA“ verwendet, welches eine Spezifikation für JPA (Java Persistence Layer) darstellt. Hibernate wird als Implementierung für das ORM-Framework verwendet und direkt von Spring-Data genutzt.

Im RDBMS werden bspw. die **Spielhistorie** persistiert, welche das Datum des jeweiligen Spiels, die Spiel-Teilnehmer sowie deren Highscores enthält. Des Weiteren werden auch die **Benutzerdaten** welche bei der **Registrierung** vergeben wurden in der Datenbank gespeichert.

Ein weiterer Vorteil der H2 Database Engine ist die in der Software-Bibliothek enthaltene Weboberfläche zur Verwaltung von Datenbanken, sodass auch die Installation und Konfiguration eines Datenbank-Management-Tools entfällt:

### API-Layer - HTTP-Schnittstelle / RESTful Webservice

Der API-Layer definiert die vom Backend bereitgestellten Schnittstellen welche zur Kommunikation / dem Datenaustausch vom Frontend mit dem Backend bereitgestellt werden. Die Schnittstelle des Backend basiert auf dem HTTP (Hypertext Transfer Protocol) und stellt eine REST-Schnittstelle dar (Representational State Transfer). Dies bedeutet, dass jeder HTTP-fähige Client die Schnittstelle des Backend konsumieren (z.B. auch Internetbrowser, cURL, etc.) kann.

Das Front-End *SnakeFX* verwendet die Java-Bibliothek *Unirest* für die Kommunikation zwischen Front- und Backend. Die Funktionsweise einer REST-Schnittstelle basiert auf dem Gedanken die grundlegenden Operationen des HTTP – wie z.B. GET, PUT, POST, DELETE, ... - auf Endpunkte / URLs eines Systems abzubilden. Dabei soll eine HTTP-GET Anfrage (Request) nur für den Bezug von Daten zuständig sein – ein HTTP-POST oder -PUT Request hingegen für die Entgegennahme neuer Daten.

### HTTP-Endpoints

Die Schnittstelle bietet folgende URLs für die Kommunikation über HTTP mit Clients an:

URL	HTTP-Methode	Beschreibung
http://localhost:8080/api/login	POST	Liefert einen JSON-Web-Token zurück welcher für die Kommunikation mit anderen Endpoints genutzt werden kann
http://localhost:8080/api/register	POST	Registriert einen Spieler mit einem Benutzernamen und Passwort
http://localhost:8080/spieler/	GET	Gibt eine Liste alle Spieler als JSON zurück
http://localhost:8080/spiele	GET	Gibt eine Liste aller Spiele als JSON zurück
...	...	...



## STOMP-Endpoints

Die folgenden URLs dienen als Adresse / Endpoints für die Kommunikation mittels WebSockets

Endpoint	Beschreibung
ws://localhost:13373/snakeserver	HTTP-Handshake und WebSocket / STOMP Upgrade
ws://localhost:13373/app/games/{gameId}	Übertragung von Spielerdaten an das Backend. Empfangene Daten werden an alle verbundenen Clients repliziert.
ws://localhost:13373/app/games/{gameId}/{playerId}	Bekanntgabe von Spielern aus der Lobby welche einem Spiel beitreten
ws://localhost:13373/app/games/	Veröffentlichung neuer Spiel-Daten (z.B. Spiel „TestSpiel“ mit max. 4 Spielern, 10 gleichzeitigen PowerUps, usw.)
ws://localhost:13373/app/players/	Bekanntgabe des Beitritts von Spielern in die Lobby

## Communication / Information-Transport

Der Communications-Layer definiert die auf dem STOMP-Protokoll / WebSockets basierende Kommunikation vom Backend mit Clients. Die ausgewählte Implementierung für das STOMP-Protokoll / WebSockets entstammt dem Spring-Messaging Projekt.

Der folgende Quelltext-Auszug des StompServiceControllers innerhalb des Backends verdeutlicht die Funktionsweise und die Implementierung des STOMP-Protokolls:

```
@Messaging("/games/{gameId}/{playerId}")
@SendTo("/topic/games/{gameId}/{playerId}")
public PlayerJoinsGameMessage broadcastPlayerJoinedGameToClients(
    @DestinationVariable String gameId,
    @DestinationVariable String playerId,
    PlayerJoinsGameMessage message)
{
    System.out.println("Player " + playerId + " joins the game " + gameId);

    // make the player join the lobby
    RunningGame destination = null;

    for (RunningGame runningGame : lobbyController.getRunningGames()) {
        if (runningGame.getStompPath().equals(message.gameToJoin.stompPath)) {
            destination = runningGame;
        }
    }

    List<Spieler> newActivePlayers = new LinkedList<>(destination.activeClients);
    newActivePlayers.add(message.spieler);
    destination.activeClients = newActivePlayers;
    message.allGames = lobbyController.getRunningGames();

    return message;
}
```

Sobald ein Client (Anwender) mittels StompClient eine Nachricht an das Backend (SnakeServer) an die URL `ws://localhost:13373/app/games/{gameId}/{playerId}` sendet (bspw. mit dem Funktionsaufruf



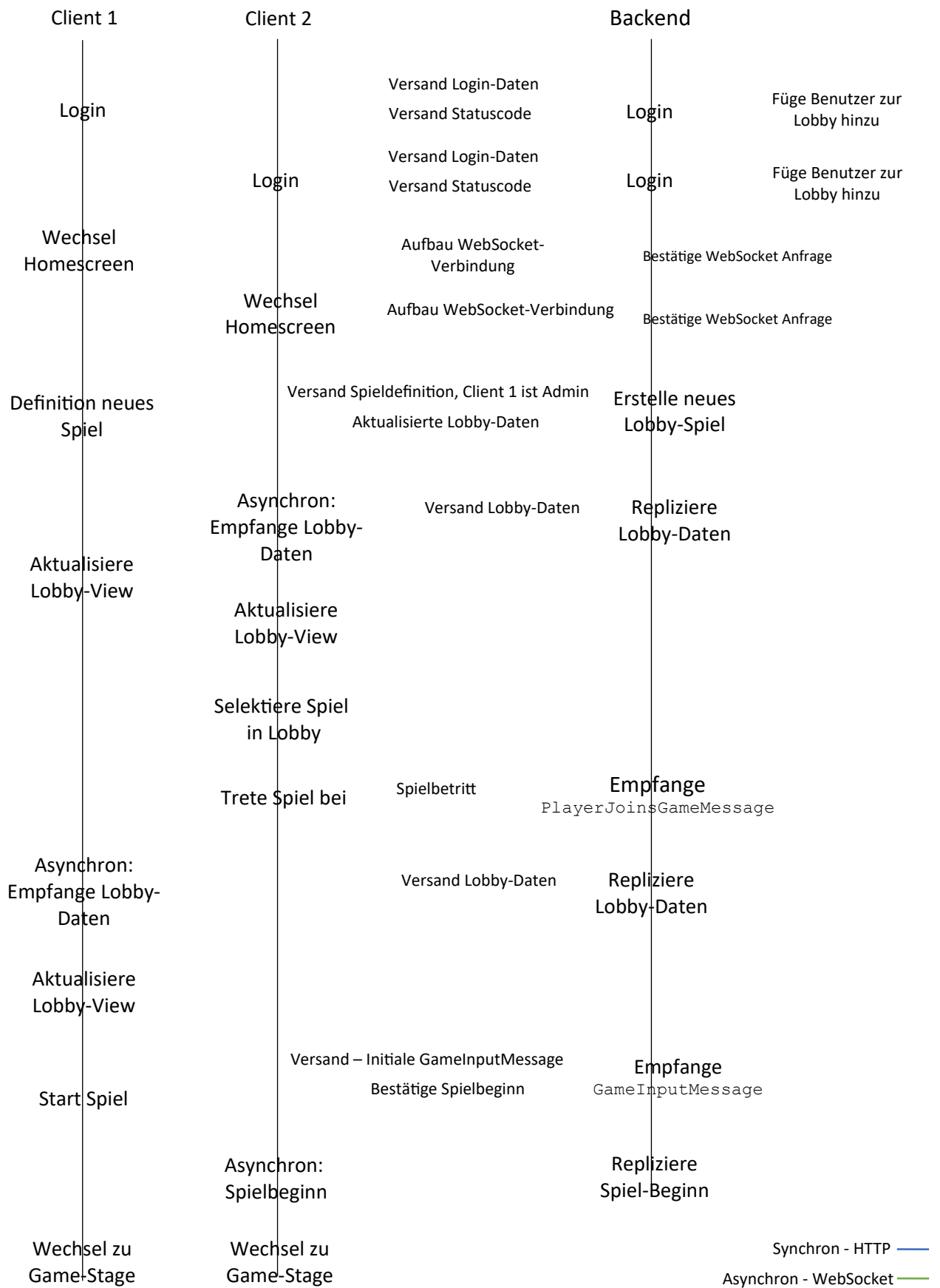
*getApplication().getStompClient().sendMessage(<topic>, <message-Instanz>))* wird die jeweilige annotierte Methode im Backend aufgerufen.

Im Fall der zuvor genannten URL dient die Funktion die Spieler-Daten welche in der Message-Instanz (PlayerJoinsGameMessage) enthalten sind zu empfangen.

Im Anschluss dessen wird überprüft ob bereits ein Spiel in der Lobby, zu welchem der Spieler beitreten möchte, vorhanden ist – ist dies der Fall wird der Spieler der Liste der aktiven Teilnehmer hinzugefügt.

Die letzte Zeile „return message;“ realisiert die Replizierung der Daten vom Backend an alle verbunden Clients / Teilnehmer (mittels MessageBroker) sodass die aktualisierte Liste der aktiven Teilnehmer wiederum bei jedem Client (mittels entsprechender Listener und Callback-Mechanismen) in Echtzeit aktualisiert wird – so hat jeder Client die Möglichkeit nachzuverfolgen welcher Spieler an einem Spiel in der Lobby teilnimmt.

Das folgende Sequenzdiagramm stellt die Kommunikation zwischen Clients und Backend während des Login-Prozesses inklusive der Erstellung eines Spiels durch einen Client und der Teilnahme eines anderen Clients am jeweiligen Spiel sowie den letztendlichen Spielstart dar.



## Umsetzung der Anforderungen

Die Inhalte des Kapitels beschreiben die technischen Implementierungen für die fachlichen Anforderungen des Projekts.

### Accountverwaltung

Das Backend besitzt einen Controller welcher einen (RESTful) Webservice für Account-Daten bereitstellt. Das Frontend hingegen besitzt einen HTTP-Client womit HTTP-Requests versendet und HTTP-Responses empfangen werden können.

Sobald ein Spieler im Login-Screen des Front-Ends einen Benutzernamen und ein Passwort einträgt und die Schaltfläche „Registrieren“ betätigt sendet der im Front-End enthaltene HTTP-Client einen POST-Request an die HTTP-Schnittstelle des Backends (URL: <http://localhost:13373/api/register/>).

Als Header-Daten des Post-Requests werden der Benutzername sowie dessen Passwort versendet. Diese Daten werden im Backend empfangen und mittels Unmarshalling vom Backend in eine Instanz der Spieler-Klasse konvertiert und in der Datenbank mittels ORM persistiert. Die Nutzerdaten des registrierten Benutzers können anschließend für die Anmeldung am System verwendet werden.

Der Login-Prozess erfolgt analog zum Registrierungsprozess – nur dass der Spieler die „Login“ Schaltfläche betätigen muss. Nach Empfang der Login-Daten wird im Backend innerhalb der ApiControllerer.login()-Methode überprüft ob ein Benutzer mit den jeweiligen Daten vorhanden ist – ist dies der Fall so wird ein gültiger Statuscode (200 = OK) an den Client zurückgegeben und der Wechsel in den Homescreen durchgeführt – bei invaliden Login-Daten wird der Statuscode 400 = BadRequest zurückgegeben ohne weitere Konsequenzen.

### Spielhistorie

Sobald ein Anwender den Spielhistorie-Screen aufruft wird ein HTTP-GET-Request an das Backend gesendet – daraufhin wird im Backend die Methode ApiControllerer.getHistorie() aufgerufen welche wiederum alle Datensätze des Typs „Spielstands“ innerhalb des RDBMS bezieht und mittels Spring Web in einen JSON-String konvertiert.

Als Antwort an den GET-Request wird eine HTTP-Response erzeugt und an den anfragenden Client gesendet welche wiederum den jeweiligen JSON-String enthält.

Sobald der Client die Antwort auf den GET-Request erhält werden die empfangenen JSON-Daten konvertiert und entsprechend einer ListView dargestellt.

### Erstellung und Wiedergabe von Sound-Effekten

Die im Front-End verwendeten Sound-Effekte wurden mit der Open-Source Software **SFXR** erstellt. Im SnakeFX-Modul ist die Klasse SoundManager enthalten welche eine Implementierung des Singleton-Entwurfsmusters darstellt. Soll ein Sound-Effekt als Reaktion einer bestimmten Spielaktion oder durch Betätigung einer Schaltfläche durchgeführt werden können die verschiedenen Play\*-Funktionen des SoundManagers verwendet werden.

Sammelt ein Spieler während eines aktiven Snake-Spiels bspw. ein Food-Element ein so wird die Funktion: `getApplication().getSoundManager().playPickup()` aufgerufen und somit die Datei `pickup.wav` abgespielt.

## Entwurfsmuster

Das Kapitel stellt dar in wie fern die verschiedenen Entwurfsmuster implementiert und genutzt werden.

### Factory-Pattern

Das Factory-Pattern wird verwendet um Instanzen von Food-Elementen und Power-Ups zu erzeugen. Dazu wurde eine `MapEntityFactory`-Klasse implementiert welche verschiedene Arten von `MapEntity` ableitenden Klassen erzeugt. Diese Instanzen werden auf Basis von lokal- und vom Backend-erzeugten Daten generiert – bspw. wird die Position jeder `MapEntity` vom Server berechnet und an die Factory weitergegeben. Zur Realisierung von `MapEntity`-Effekten wird eine Integer-Klassenvariable definiert anhand dessen entschieden wird um welche konkrete Effekt-Art es sich bei der `MapEntity`-Instanz handelt – dabei wurden folgende Power-Ups realisiert.

MapEntity-Typ	Auswirkung / Effekt
FoodEntity	ID = 1 – Erhöht den Spieler-Score
PredatorEntity	ID = 2 – Der Konsum ermöglicht die Aktivierung der Predator-Mechanik welche ermöglicht Schlangenkörper-Elemente anderer Spieler abzubeißen
FreezeEntity	ID = 3 – Stoppt die Bewegung aller anderen Spieler

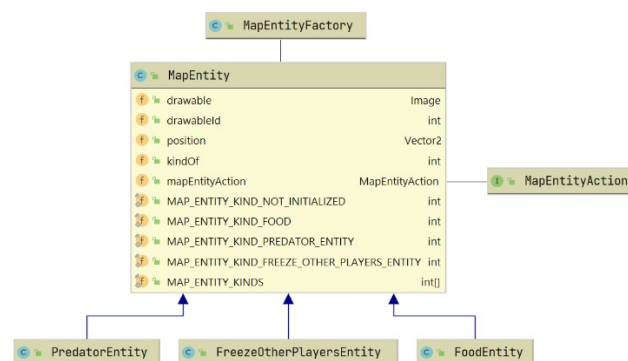


Abbildung 9 - Integration der `MapEntityFactory` und die dadurch erzeugten `MapEntity`-Instanzen

### Observer-Pattern

Das Observer-Pattern wird verwendet um verschiedene Zustände während des Spiels zu beobachten und entsprechend zu reagieren. Beispielsweise wird mithilfe des Observer-Patterns geprüft ob ein Spieler eine Spiel-Beendende Reaktion auslöst und somit als aktiver Teilnehmer ausscheidet. Des Weiteren werden verschiedene im JavaFX-Framework bereits enthaltene Observer-Implementierungen verwendet um Zustände von UI-Elementen zu überwachen und zu kontrollieren – bspw. wenn die Stage der Anwendung geschlossen wird um zuvor einen „Möchten Sie sich wirklich abmelden“-Dialog anzeigen zu können – oder um auf eingehende STOMP-Nachrichten zu reagieren wenn in der Lobby neue Spieldaten angezeigt werden sollen.

### Composite-Pattern

Das Composite-Pattern wird ebenfalls im GameController eingesetzt um Typen-unabhängige Effekte von `MapEntity`-Entitäten zu realisieren und auszulösen. Zur Ausführung eines Power-Up-Effekts reicht der Aufruf der Methode:

```
mapEntity.mapEntityAction.onExecute(snakePlayerMap.get(snake), snake, this.runningGame);
```

## Erweiterungen und Extras

Die folgenden Erweiterungen und Extras wurden im Zuge des Projekts realisiert.

### Spiel- und Spieler-Management / Lobby-Metapher

Als essentieller Bestandteil einer Mehrspieler-Anwendung wurde die Anforderung identifiziert Benutzern die Möglichkeit zu geben eine einfache Auskunft über weitere angemeldete Benutzer zu erhalten sowie Informationen über startende sowie bereits laufende Spielrunden zu beziehen.

Aus diesem Grund wurde eine „Lobby“-Mechanik realisiert. Dies bedeutet, dass alle Spieler nach der Anmeldung (Login) am System einer sogenannten „Lobby“ (einem Empfangsraum) beitreten. Das Front-End besitzt innerhalb der Homescreen-Oberfläche eine ListView in dem die Spieler der Lobby dargestellt werden.

Dieses Vorgehen bietet für Benutzer den Vorteil eine Auskunft über alle derzeit angemeldeten Benutzer zu erhalten. So ist ein Spieler bspw. in der Lage zu identifizieren ob ein Teilnehmer für den Start einer zwei-Spieler-Partie des Snake-Spiels vorhanden ist.

### Chat-System / Diagnostik zwischen Client und Server

Als weiterer Bestandteil einer Mehrspieler-Anwendung wurde die Anforderung identifiziert, dass zusätzlich zum Lobby-System eine Kommunikations-Möglichkeit für Benutzer vorhanden sein sollte, sodass sich Spieler untereinander abstimmen können um Faktoren eines startenden Spiels miteinander abzusprechen oder einfach nur über allgemeine Themen diskutieren können.

Des Weiteren wurde ein Bot-User (ein leichtgewichtiges Antwort-System) im Chat integriert, sodass Spieler zusätzliche Informationen in Form einfacher Fragen formulieren können und dadurch automatisch Informationen zu verschiedenen Faktoren erhalten können.

So kann bspw. ein Benutzer einen Satz wie „Whats the time?“ mittels Chat an das Backend übertragen womit die aktuelle Uhrzeit vom Bot-User wiedergegeben wird.

Ein weiteres Beispiel dieser Funktionalität stellt der Sachverhalt dar, dass ein Benutzer das Wort „ping“ mittels Chat an das Backend übertragen kann worauf der Bot-User mit dem Wort „pong“ antwortet – dies stellt eine Analogie aus traditionellen Netzwerken dar in dem das Diagnosewerkzeug „ping“ eine ICMP-Anfrage in ein IP-Netzwerk schickt und von evtl. vorhandenen Netzwerk-Clients eine ICMP-Antwort „pong“ erhält – diese Funktionalität ist also auch als Diagnosewerkzeug geeignet um den korrekten Ablauf der Kommunikation zwischen Client und Server sicherstellen zu können.

## Nicht realisierte Anforderungen

Aufgrund des Projekt-Umfangs und der Tatsache, dass nur zwei anstelle von drei bis vier Teilnehmern, die Projekt-Gruppe definieren wurde die Anforderung „JSON Web Tokens zur Authentifizierung von Anwendern“ nicht realisiert – da das Anwendungssystem jedoch Spring sowie verschiedene Module des Spring-Ökosystems verwendet und das Spring-Framework mit Spring-Security ein Modul zur Absicherung diverser Ressourcen (wie RESTful Webservices, HTTP-Kommunikation, etc.) inklusive verschiedene Authentifizierung-Mechanismen unterstützt wäre die

technische Basis für die Realisierung der Absicherung und den Austausch von JSON Web Tokens bereits umgesetzt.

## Bedienungsanleitung und Spielregeln

Die folgende Abschnitte des Kapitels erläutern verschiedene Aspekte rund um die Einrichtung, die Verwendung und die Details des Anwendungs-Systems.

### Einrichtung

Nach dem Bezug des Workspaces von GitHub oder der Software-Artefakte innerhalb der Abgabe muss zuerst das Backend „SnakeServer“ gestartet werden. Da alle benötigten Services Teil des Workspaces und somit auch des Artefakts sind ist keine weitere Installations- oder Konfigurations-Arbeit notwendig.

Das Backend wird mit folgendem Befehl gestartet: `java -jar SnakeServer-0.0.1-SNAPSHOT.jar`

Oder mit der Ausführung der `main`-Methode der Klasse `SnakeServerApplication`.

Das Front-End wird mit folgendem Befehl gestartet: `java -jar snakefx-0.0.1-SNAPSHOT.jar`

Oder mit der Ausführung der `main`-Methode der Klasse `AppSnakeFX`.

*Eine ausführlich bebilderte Beschreibung wie der Workspace innerhalb der IDE IntelliJ IDEA eingerichtet werden kann befindet sich unter: [https://github.com/Bummelnderboris/Patterns\\_and\\_Frameworks/tree/master/docs/Setup\\_Project\\_And\\_Execute\\_it](https://github.com/Bummelnderboris/Patterns_and_Frameworks/tree/master/docs/Setup_Project_And_Execute_it)*

### Anmeldung / Registrierung

Nach dem Start des Front-Ends wird dem Anwender der Login-Screen präsentiert:

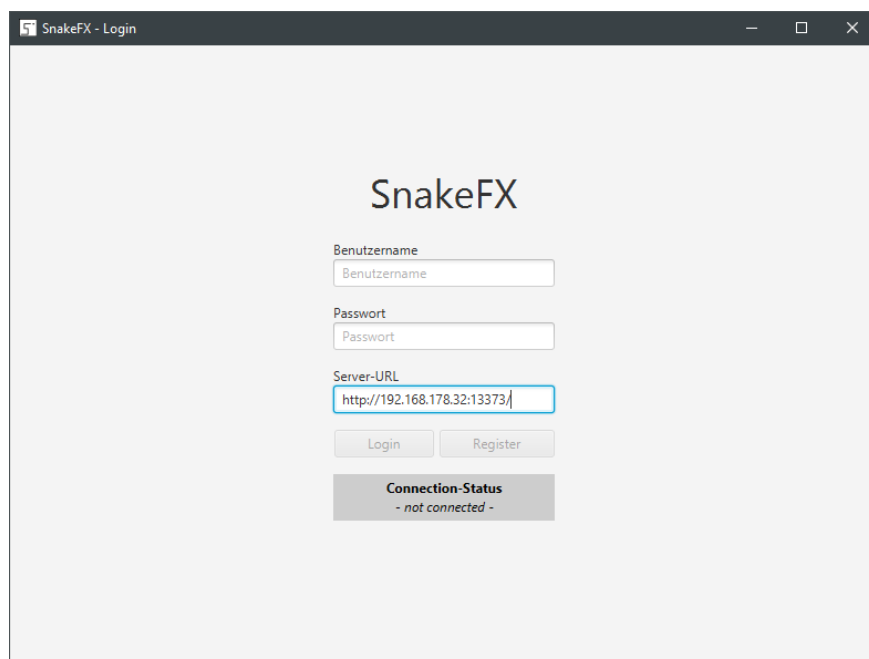


Abbildung 10 – Login-Screen des Front-Ends



Zur Anmeldung am Backend sind Benutzerdaten (Benutzername + Passwort) erforderlich – dazu können entweder die hartkodierte Test-Daten („benni“, „WiSe2020!“) verwendet oder neue Benutzerdaten erstellt werden. Für die Erzeugung eines neuen Benutzerkontos können in den Texteingabe-Schaltflächen jeweils Benutzername- und Passwort eingetragen werden – durch Betätigung der „Register“-Schaltfläche wird eine Anfrage an das Backend übertragen und ein neues Benutzerkonto erstellt.

## Homescreen – Lobby, Chat-System, Spiel-Management, Spielhistorie

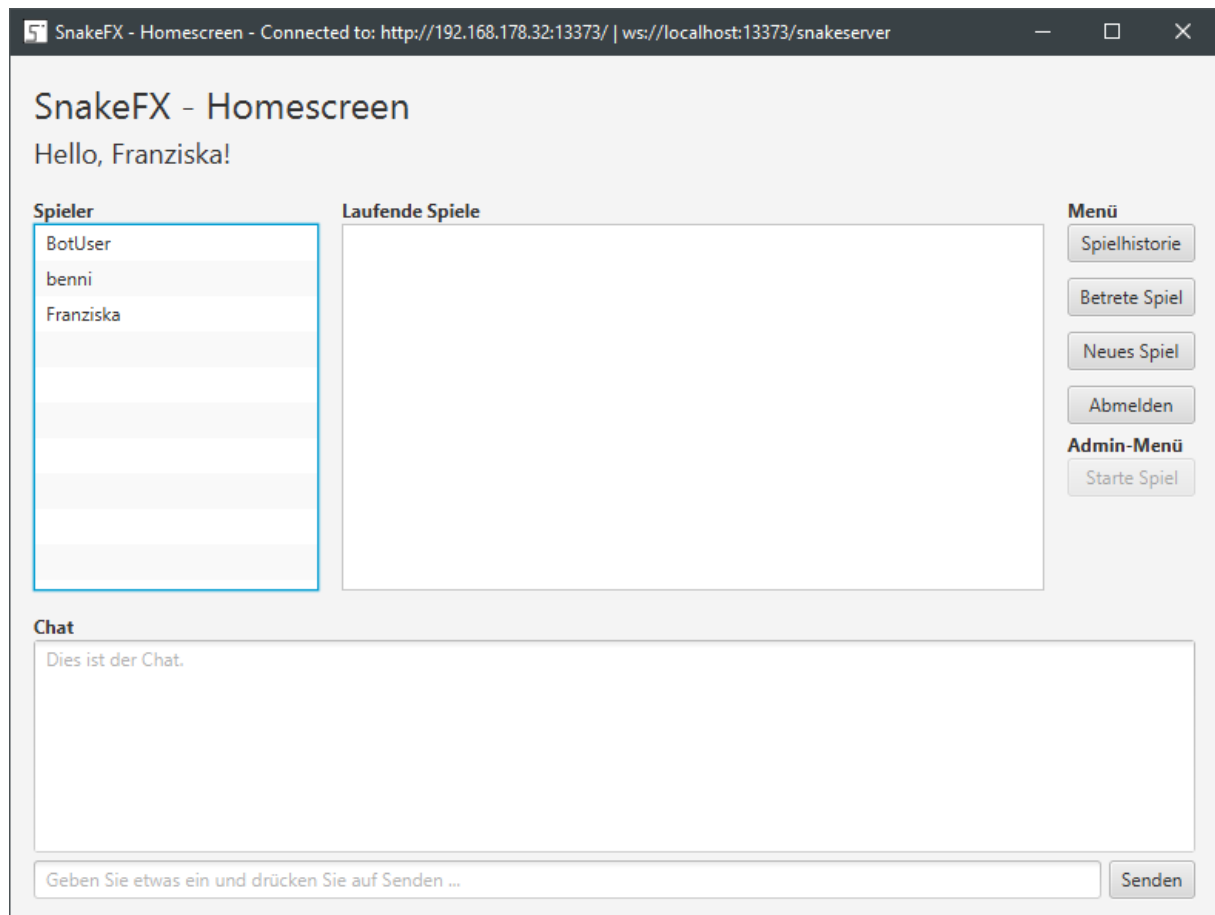
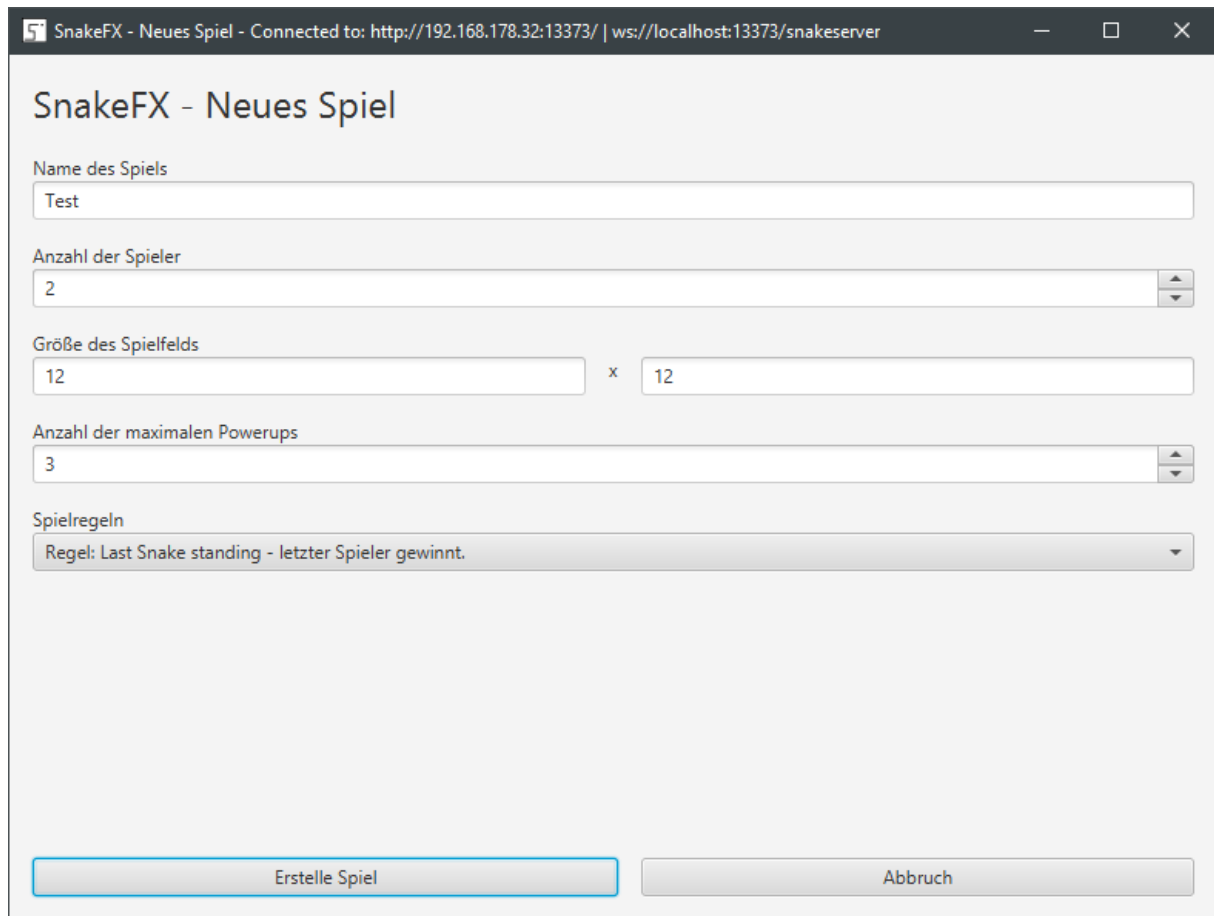


Abbildung 11 - Home-Screen des Front-Ends

Der Home-Screen stellt die Lobby und dessen fachliche Elemente visuell dar. Auf der linken Seite befinden sich alle zum Zeitpunkt der Betrachtung angemeldeten Benutzer / Clients. Im Zentrum befindet sich eine Auflistung aller zu startenden / laufenden Spiele. Im Unteren Bereich der Abbildung ist das Chat-System ersichtlich welches für die Kommunikation zwischen den Teilnehmern verwendet werden kann.

Möchte ein Benutzer ein neues Spiel erstellen so muss dieser die Schaltfläche „Neues Spiel“ betätigen – daraufhin wird im Front-End auf die Benutzeroberfläche zur Erstellung eines neuen Spiels gestartet und angezeigt:



The screenshot shows a web application window titled "SnakeFX - Neues Spiel" with a status bar indicating a connection to "http://192.168.178.32:13373/" and "ws://localhost:13373/snakeserver". The main content area contains several configuration fields: "Name des Spiels" with the value "Test", "Anzahl der Spieler" with the value "2", "Größe des Spielfelds" with two input fields containing "12" and "12" separated by an "x", "Anzahl der maximalen Powerups" with the value "3", and "Spielregeln" with a dropdown menu showing "Regel: Last Snake standing - letzter Spieler gewinnt.". At the bottom, there are two buttons: "Erstelle Spiel" and "Abbruch".

Abbildung 12 - New-Game-Screen des Front-Ends

SnakeFX & SnakeServer ermöglichen eine feingranulare Definition von verschiedensten Spiel-Szenarien. Beispielsweise können die Anzahl der maximalen Teilnehmer definiert werden. Des Weiteren kann die Größe des Spielfelds sowie die maximale Anzahl an simultan sich auf dem Spielfeld befindlichen Power-Ups konfiguriert werden. Das letzte Konfigurationselement stellt die Auswahl der Spielregeln dar – hier kann ausgewählt werden ob eine bestimmte Punktzahl erreicht werden muss oder ob der letzte überlebende Spieler als Gewinner zählt. Aufgrund des flexiblen Entwicklungsmodells könnten verschiedene weitere Spielregeln erdacht und implementiert werden.

Betätigt der jeweilige Benutzer die „Erstelle Spiel“-Schaltfläche so wird ein neues Spiel in der Lobby veröffentlicht und in allen verbundenen Clients mittels WebSocket / STOMP – Nachricht bekanntgegeben und die jeweiligen Front-Ends aktualisiert.

Der Benutzer welcher das Spiel erstellt wird als Admin / GameMaster gekennzeichnet und ist als einziger Benutzer im Stande das jeweilige erstellte Spiel zu starten. Alle weiteren Benutzer können über die Schaltfläche „Betrete Spiel“ am jeweiligen Spiel teilnehmen. Startet der Admin / GameMaster das Spiel werden alle Front-End-Instanzen in die Game-Screen Oberfläche gewechselt worin das jeweilige Spiel visualisiert wird.

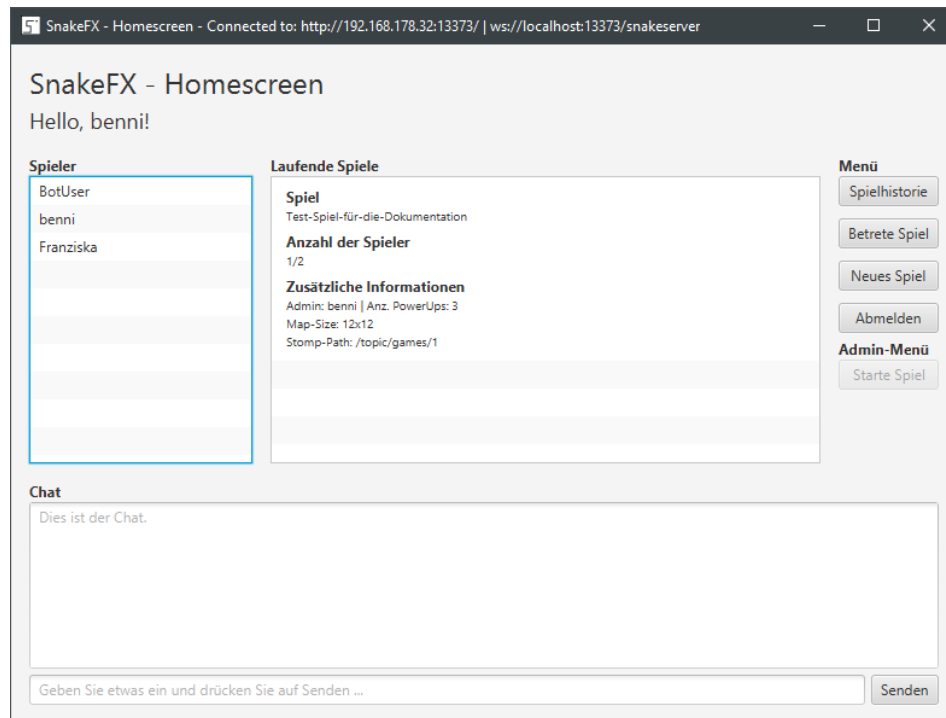


Abbildung 13 - Aktualisierte Lobby mit einem startenden Spiel

Möchte ein Benutzer alle im Backend persistierten Spiele und deren konkrete Werte betrachten so muss dieser im Homescreen die Schaltfläche „Spielhistorie“ betätigen – daraufhin wechselt das Front-End in den Game-History-Screen in der alle absolvierten Spiele inkl. der Teilnehmer sowie deren Highscores visualisiert werden:

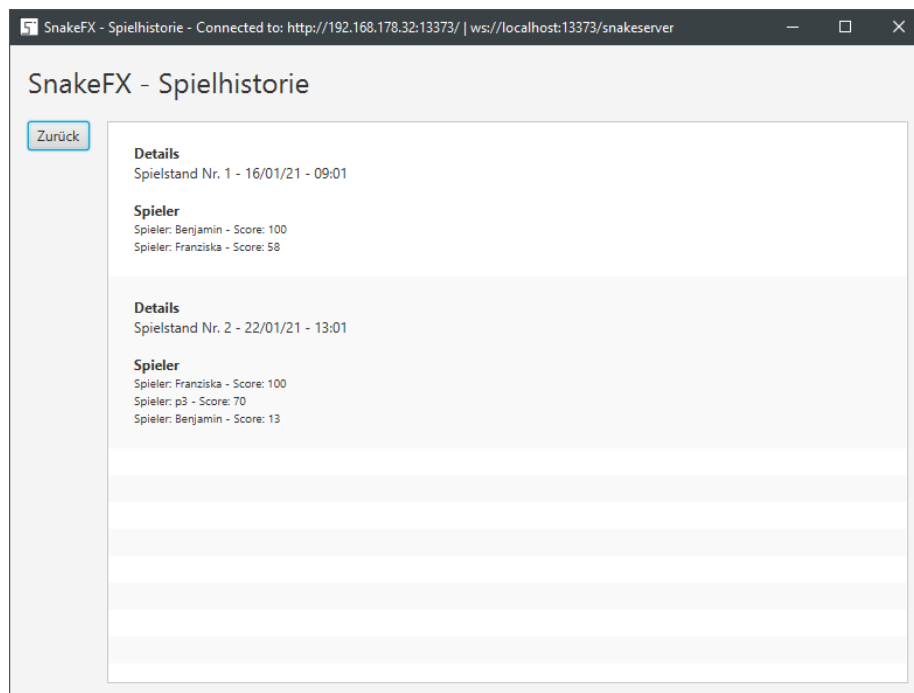


Abbildung 14 - Benutzeroberfläche der Spielhistorie im Front-End

## Game-Screen

Nach der Erzeugung eines Spiels innerhalb der Lobby, der Teilnahme weiterer Benutzer am jeweiligen Spiel und des Spiel-Starts durch den GameMaster wechselt das Front-End in den Game-Screen.

In dieser Benutzeroberfläche stellt das Front-End den aktuellen Zustand eines laufenden Spiels dar.

Je nach Spieldefinition (Konfiguration innerhalb des New-Game-Screens) werden verschieden viele Teilnehmer sowie Food-Elemente & Power-Ups dargestellt.

Zur Steuerung der Schlange können folgende Tasten verwendet werden:

Taste	Alternative Taste	Aktion
W	Pfeiltaste Hoch	Bewege Schlange nach oben
A	Pfeiltaste Links	Bewege Schlange nach links
S	Pfeiltaste Runter	Bewege Schlange nach unten
D	Pfeiltaste Rechts	Bewege Schlange nach rechts
R	Leertaste	Stoppe Schlange an der aktuellen Position



Abbildung 15 - Prototypische Darstellung des Game-Screens

Löst ein Spieler eine Bedingung / Aktion aus welche zum Spiel-Ende führt so wird dies mit der Einblendung des „Game Over“-Screens visualisiert. Der Spieler kann dann nur noch passiv am Spiel teilnehmen und die weiteren Züge der Mitspieler betrachten – selbst jedoch keine weiteren Eingaben tätigen. Durch die Betätigung der „Show Homescreen“-Schaltfläche gelangt ein Benutzer wieder zurück zum Homescreen in dem neue Spiele erstellt oder an startenden Spiele teilgenommen werden kann.

## Projektplan

Zum Zeitpunkt der Abgabe des Projektberichts weißt das GitHub-Repository über 135 [Commits](#) auf.

Beteiligte Gruppenmitglieder	Erledigte Aufgaben / Aspekte
Gesamtes Team	Erfassung der Anforderungen
Benjamin Wulfert	Entwurf der UML-Diagramme
Leonard Reidel	Ergänzung der UML-Diagramme
Benjamin Wulfert	Erstellung ScreenFlow / WireFrames für Front-End Kompletter Entwurf, erste Variante des Front-Ends inkl. Wechsel der Benutzeroberflächen ( <i>Was passiert wenn ich auf Login, Spielhistorie, Starte Spiel, etc. klicke?</i> )
Benjamin Wulfert	Vorschlag für Ablauf und Realisierung der Spieler-Bewegung Mithilfe von linearer Algebra / Vektorrechnung
Benjamin Wulfert	Initialer Setup von Client- und Server-Projekt Konfiguration des Build-Management-Systems - Apache Maven
Gesamtes Team	Überarbeitung der UML-Diagramme
Benjamin Wulfert	Implementierung erster Entwurf aller Benutzeroberflächen des Front-Ends
Benjamin Wulfert	Erstellung d. Gantt-Diagramms, stetige Aktualisierung
Benjamin Wulfert	Backend: Realisierung des Persistence-Layers, Integration ORM-Framework
Benjamin Wulfert	Backend: Realisierung des API-Layers, Integration RESTful Web Services, JSON-Marshalling
Benjamin Wulfert	Entwurf Dokumentation, IDE-Setup, Dokumentation des Workspaces
Benjamin Wulfert	Front-End: Integration nebenläufiger Ausführung (Threads)
Benjamin Wulfert	Front-End: Konsum der Schnittstellen von Backend durch Front-End (Bezug der JSON-Daten von RESTful Webservices)
Benjamin Wulfert	Aktualisierung UML-Diagramme, Gantt-Diagramme
Benjamin Wulfert	Aktualisierung der Dokumentation
Benjamin Wulfert	Back-End: Integration relationaler Daten-Modelle (ORM)
Leonard Reidel	Entwurf Schnittstellenbeschreibung
Benjamin Wulfert	Separation von Kern-Logik & Modellen aus Front-End, Etablierung Core-Projekt
Benjamin Wulfert	Entwicklung von Login- und Registrierungs-Prozess sowohl im Front-End als auch im Backend
Benjamin Wulfert	Refactoring des Clients – Entwicklung BaseController und BaseApplication
Leonard Reidel	Entwurf für Standalone Singleplayer Snake-Spiel
Benjamin Wulfert	Integration Standalone Singleplayer Snake-Spiel in Front-End
Benjamin Wulfert	Erweiterung Snake-Spiel: Mehrere steuerbare Schlangen
Benjamin Wulfert	Erweiterung Snake-Spiel: NPC-Fähigkeit für Test der Snake-Logik Aktualisierung der HTTP-Schnittstellen
Benjamin Wulfert	Aktualisierung der Dokumentation
Benjamin Wulfert	Entwurf Lösungsdokument für Abgabe V1, Abgabe
Leonard Reidel	Aktualisierung Snake-Spiel (Schlange konsumiert Element von Schlange)
Benjamin Wulfert	Update der Lösungsdokumente, Entwurf Handout
Leonard Reidel	Aktualisierung Gantt-Diagramm
Benjamin Wulfert	Update der Lösungsdokumente, Entwurf der Verwendung von Design-Patterns
Benjamin Wulfert	Front-End: Stage-Wechsel im selben Fenster Aktualisierung Lösungsdokumente V2, Berücksichtigung des Team-Feedbacks, Abgabe
Benjamin Wulfert	Front-End: Anbindung von Backend an Front-End: - Datenaustausch beim Login / bei der Registrierung - Datenaustausch und Visualisierung der Lobby-Daten für aktive Teilnehmer, startende oder laufende Spiele - Datenaustausch und Visualisierung der Spielhistorie-Daten
Benjamin Wulfert	Integration von WebSockets / STOMP-Service im Backend, Implementierung von WebSocket/STOMP-Client im Front-End: - Nach Login eines Clients wird WebSocket-Verbindung zum Backend aufgebaut - Lobby-Daten werden über WebSockets/STOMP übertragen - Aktualisierung ORM / Persistence-Layer
Benjamin Wulfert	Refactoring Lobby-System: Lobby-Daten (Spieler, Spiele) werden nun über WebSockets / STOMP übertragen, nur der erste Bezug erfolgt über HTTP
Benjamin Wulfert	Abschluss Lobby-System:

	<ul style="list-style-type: none"> <li>- Bei Erstellung eines Spiels wird dies an das Backend übertragen und an alle angemeldeten Clients repliziert.</li> <li>- Spiel-erstellender Spieler wird als Game-Master deklariert</li> <li>- Weitere Spieler können an zu startendem Spiel teilnehmen (join-Mechanismus)</li> <li>- Game-Master kann Spiel starten</li> <li>- Wechsel in den Game-Screen</li> </ul>
Benjamin Wulfert	Integration des Multiplayer-Aspekts in Singleplayer-Snake: - Eingabe und Versand der Spieler-Bewegungsdaten - Replikation an Clients durch Backend - Serverseitige Berechnung kollisionsfreier Positionen für Food-Elemente und Power-Ups
Benjamin Wulfert	Entwurf für Präsentation, Aktualisierung der Dokumentation
Benjamin Wulfert	Überarbeitung: - Synchronisierung der Food-Elemente und Power-Ups - Korrekte Synchronisierung der Spieler-Positionen
Benjamin Wulfert	Aktualisierung der Dokumentation Integration von Test-Projekt „SnakeTest“ zur teil-automatisierten Ausführung von Server- und Client-Startup, Entwicklung verschiedener Szenarien zur Simulation zweier Clients welche sowohl das Chat-System testen als auch ein Spiel erstellen, daran teilnehmen und jeweiliges Spiel starten.
Leonard Reidel	Integration: Schlange isst Element von Schlange-Anforderung Test des Front-Ends, Test des Backends, Kommunikation der Ergebnisse
Benjamin Wulfert	Aktualisierung der Dokumentation, Lösungsdokumente, Präsentation
Benjamin Wulfert	<ul style="list-style-type: none"> <li>- Implementierung des SoundManagers zur Wiedergabe von Sound-Effekten</li> <li>- Erstellung von Sound-Effekten mittels SFXR</li> </ul>
Benjamin Wulfert	- Implementierung der Design-Patterns: Composite, Factory und Observer
Benjamin Wulfert	Implementierung dynamische Eingabe und systemweite Verwendung der IP-Adresse für Front- und Backend
Benjamin Wulfert	Refactoring Projekt - Reduktion der technischen Komplexität - Abbau von technischen Schulden - Entfernung oder Harmonisierung hart-kodierter Daten
Benjamin Wulfert	Entwurf des Kurzberichts, Durchführung der Kapitel: - 1. Einleitung - 2. SnakeFX Front-End - 3. SnakeServer - 4. Umsetzung der Anforderungen - 5. Erweiterungen und Extras - 6. Nicht realisierte Anforderungen - 7. Bedienungsanleitung und Spielregeln - 8. Projektplan (diese Tabelle) - 9. Verwendete Software
Leonard Reidel	Entwurf Unterkapitel für Front-End: Snake-Implementierung Erweiterung des Home-Screens – Integration vers. Dialoge
Benjamin Wulfert	Überarbeitung des Kurzberichts
Benjamin Wulfert	Implementierung asynchrones GameOver - Auf jedem Client wird nun nur GameOver angezeigt wenn der jeweilige Spieler am jeweiligen Client GameOver ist (vorher wurde bei einem einzigen GameOver eines beliebigen Spielers für alle Spieler der GameOver-Screen aktiviert) - Möglichkeit zurück zum Homescreen zu gelangen bei Spiel-Ende - Persistenz der Spiel-Ergebnisse in der Spiel-Historie bei Spiel-Ende - Aktualisierung ORM / Persistence-Layer - Aktualisierung des Kurzberichts & der noch ausstehenden Tasks
Benjamin Wulfert	Aktualisierung Unterkapitel für Front-End: Snake-Implementierung auf aktuelle Version des Anwendungs-Systems
Benjamin Wulfert	Finalisierung & Refactoring der Design-Pattern Finalisierung der MapEntity / Food-Element und Power-Up Entities Überarbeitung Kapitel Umsetzung & Anforderungen - Design-Pattern
Benjamin Wulfert	Finalisierung der Abgabe, Einreichung der Abgabe

## Resümee & Ausblick

Das Semester-Projekt für das Modul „Patterns & Frameworks“ konnte innerhalb des verfügbaren Zeitrahmens (und anforderungsgerecht) (bis auf die Integration der JSON-Web-Tokens) realisiert werden. Das Projekt wurde in einem wohl separierten Multi-Modul Ansatz realisiert in dem pro Domäne (Kern-Logik, Front-End, Back-End, Test-Projekt) ein eigenes Modul definiert wurde was die Erweiterbarkeit und Trennung von Fachlogik und technischer Implementierung erleichtert. Des Weiteren wurden alle Module im Hinblick auf zukünftige Erweiterungen realisiert sodass auch nach Abgabe des Projektberichts weitere zusätzliche Funktionalitäten realisiert werden können.

Das Backend setzt auf verschiedene Module / technische Implementierungen des Spring-Ökosystems welches als De-Facto Industriestandard für Backend-Entwicklung gilt. Des Weiteren wurde das Backend als auch das Frontend strikt nach der MVC-Architektur (Model View Controller) umgesetzt. Das Backend besteht aus verschiedenen logischen Ebenen / Layer wie bspw. dem Persistence-Layer welcher verschiedene Aufgaben bzgl. RDBMS / CRUD realisiert oder dem API-Layer welche verschiedene URLs / Endpunkte für den Konsum von RESTful Webservices bereitstellt. Die Kommunikation mittels WebSockets erfolgt mittels STOMP-Protokoll – die dazugehörigen Funktionalitäten sind im StompServiceController enthalten – das Gegenstück innerhalb des Front-Ends bildet der StompClient welcher die STOMP-Nachrichten empfangen und verarbeiten kann.

Das Frontend setzt auf JavaFX – dem aktuellsten Framework für die Entwicklung moderner Benutzeroberflächen – so ist es durch Cross-Kompilierung bspw. auch möglich einen Client für Android-Smartphones zu erzeugen und zur Kommunikation mit dem Backend zu nutzen. Die Anwendung stellt eine Single-Page-Application dar in welcher jede Ansicht in einer eigenen Scene jedoch immer im selben Fenster dargestellt wird. Jeder Screen / Scene wird mittels dazugehörigen Controller gesteuert – so gibt es bspw. für den HomeScreen einen HomeScreenController, für den LoginScreen einen LoginScreenController und für den GameScreen einen GameScreenController. Letzterer ist das zentrale Steuerelement zur Durchführung der Fachlogik so wie der Darstellung des Spielzustandes. Mittels StompClient werden Spielereingaben an das Backend übertragen und von dort aus an alle am Spiel teilnehmenden Spieler repliziert. Die Berechnung der Positionen und der Bewegungen geschieht mittels traditioneller Vektoraddition. Bei dem Konsum verschiedener MapEntities (Food-Elemente, Power-Ups) werden mittels SoundManager verschiedene Sound-Effekte wiedergegeben.

Das Anwendungs-System unterstützt alle im Anforderungsdokument geforderten Anforderungen (bis auf den JSON-Web-Token-Austausch) und darüber hinaus die Möglichkeit Spieler in Echtzeit miteinander kommunizieren zu lassen. Des Weiteren ermöglicht die Implementierung des Lobby-Systems eine benutzerfreundliche Variante Spiele zu definieren und für beliebige Spieler zugänglich zu machen.

## Verwendete Software

Die folgende Software wurde im Zuge der Projektdurchführung und Entwicklung verwendet.

Name / Titel	Quelle
IntelliJ IDEA	<a href="https://www.jetbrains.com/de-de/idea/">https://www.jetbrains.com/de-de/idea/</a>
Git	<a href="https://git-scm.com/">https://git-scm.com/</a>
Apache Maven	<a href="https://maven.apache.org/">https://maven.apache.org/</a>
H2 Database Engine	<a href="https://www.h2database.com/">https://www.h2database.com/</a>
Spring Boot	<a href="https://spring.io/projects/spring-boot">https://spring.io/projects/spring-boot</a>
Spring Data JPA	<a href="https://spring.io/projects/spring-data-jpa">https://spring.io/projects/spring-data-jpa</a>
Spring Messaging	<a href="https://spring.io/guides/gs/messaging-stomp-websocket/">https://spring.io/guides/gs/messaging-stomp-websocket/</a>
JavaFX	<a href="https://openjfx.io/">https://openjfx.io/</a>
Unirest	<a href="http://kong.github.io/unirest-java/">http://kong.github.io/unirest-java/</a>

SFXR	<a href="https://www.drpetter.se/project_sfyr.html">https://www.drpetter.se/project_sfyr.html</a>
------	---