Kurzbericht   
Snake-Core, SnakeFX, SnakeServer

Ostfalia Fachhochschule für angewandte Wissenschaften

Benjamin Wulfert

Leonard Reidel

Semester: Wintersemester 2020

20. Januar 2021

Inhaltsverzeichnis

[Einleitung 4](#_Toc61934902)

[Modul-Architektur 4](#_Toc61934903)

[SnakeCore 4](#_Toc61934904)

[SnakeFX - Frontend 4](#_Toc61934905)

[SnakeServer - Backend 5](#_Toc61934906)

[SnakeTest – Test-Modul 5](#_Toc61934907)

[SnakeFX - Front End 6](#_Toc61934908)

[Architektur 6](#_Toc61934909)

[BaseController 6](#_Toc61934910)

[BaseApplication 6](#_Toc61934911)

[Communication / Information-Transport 7](#_Toc61934912)

[User-Interface 8](#_Toc61934913)

[Login-Screen 8](#_Toc61934914)

[Home-Screen 8](#_Toc61934915)

[Spielhistorie-Screen 9](#_Toc61934916)

[New-Game-Screen 9](#_Toc61934917)

[Snake-Implementierung 9](#_Toc61934918)

[Schlangenexistenz und -bewegung 9](#_Toc61934919)

[Schlangeninitialisierung: 9](#_Toc61934920)

[Schlangenbewegung 10](#_Toc61934921)

[Power-Ups/Food Generierung und Essen 10](#_Toc61934922)

[Aktionen (Schlange, trifft Wand, andere Schlange, Essen, sich selbst…) 10](#_Toc61934923)

[SnakeServer 12](#_Toc61934924)

[Sourcecode-Packages 12](#_Toc61934925)

[Persistenz-Layer 12](#_Toc61934926)

[H2 Database Engine 13](#_Toc61934927)

[API-Layer - HTTP-Schnittstelle / RESTful Webservice 14](#_Toc61934928)

[STOMP-Endpoints 14](#_Toc61934929)

[Communication / Information-Transport 15](#_Toc61934930)

[Umsetzung der Anforderungen 17](#_Toc61934931)

[Accountverwaltung 17](#_Toc61934932)

[Spielhistorie 17](#_Toc61934933)

[Erstellung und Wiedergabe von Sound-Effekten 17](#_Toc61934934)

[Entwurfsmuster 18](#_Toc61934935)

[Erweiterungen und Extras 19](#_Toc61934936)

[Spiel- und Spieler-Management / Lobby-Metapher 19](#_Toc61934937)

[Chat-System / Diagnostik zwischen Client und Server 19](#_Toc61934938)

[Spieler-Teleport 19](#_Toc61934939)

[Nicht realisierte Anforderungen 19](#_Toc61934940)

[Bedienungsanleitung und Spielregeln 21](#_Toc61934941)

[Projektplan 22](#_Toc61934942)

[Verwendete Software 23](#_Toc61934943)

# 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 dieser 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 - mittels 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 Abhängigkeiten) beschreibt.

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

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

<https://github.com/Bummelnderboris/Patterns_and_Frameworks> - Leonard Reidel

### Projekt Aufteilung

Benjamin Wulfert:

Leonard Reidel: Snake Implementierung (Das Kern-Spiel)

Planung / Problemlösungen und Ideen wurden im Teams gelöst und erarbeitet.

## Modul-Architektur

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

SnakeCore

SnakeFX  
Front-End

SnakeServer  
Back-End

SnakeTest

Abbildung - 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.

## 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.

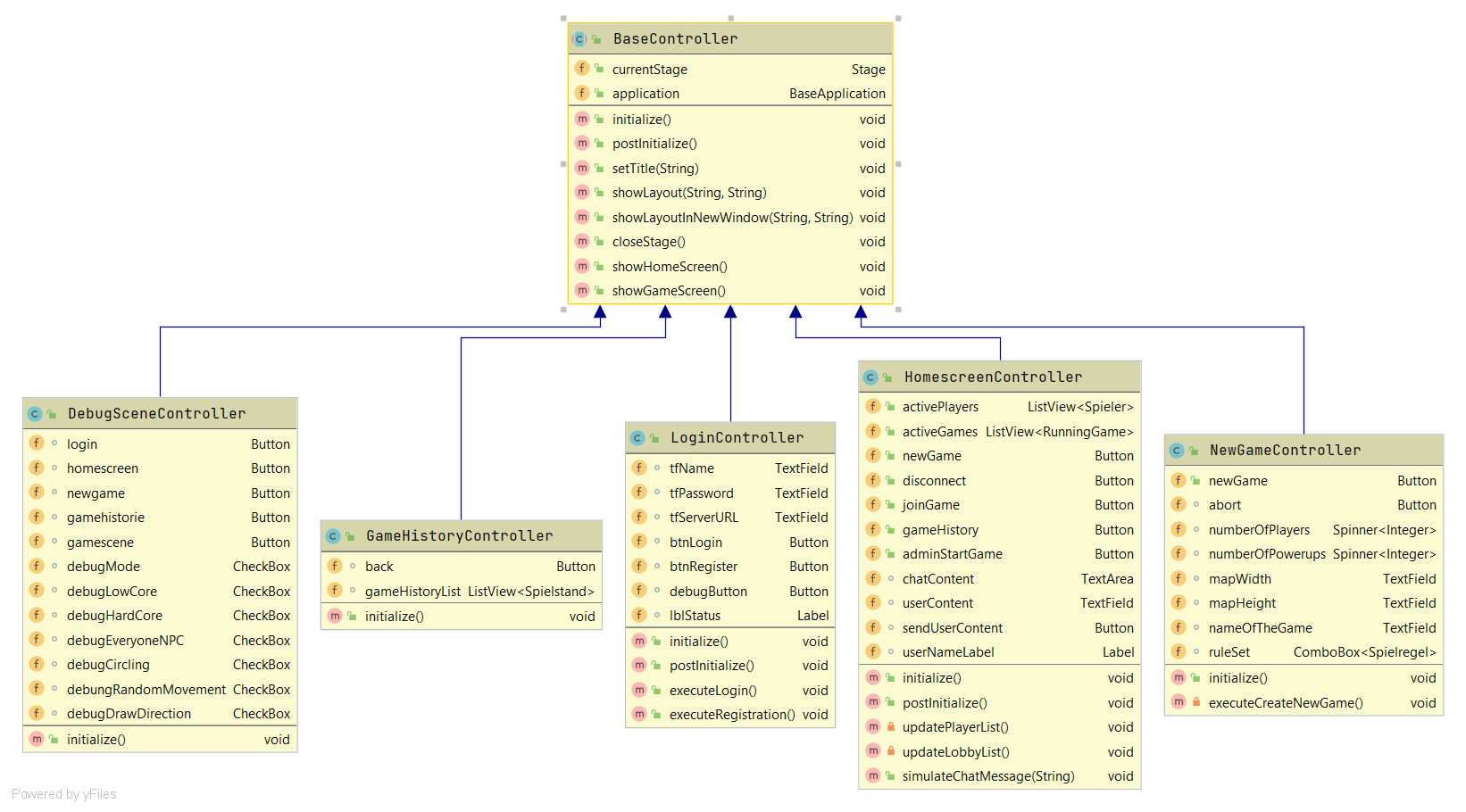


Abbildung - Klassendiagramm der implementierten Controller-Architektur des Front-Ends

### 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 von 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
* uvw.

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.

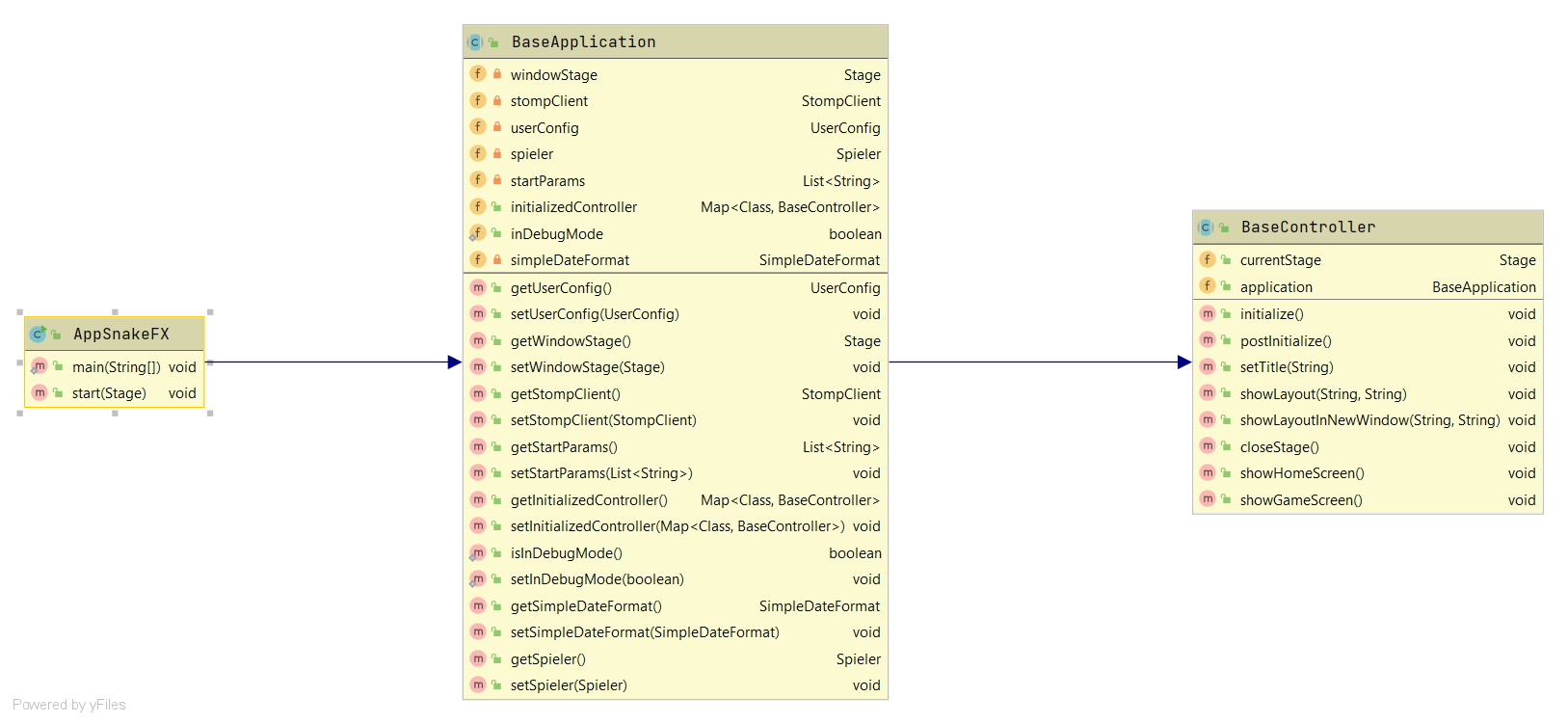


Abbildung - Klassendiagramm welches die Relation zwischen Anwendung und BaseController darstellt.

## Communication / Information-Transport

Das Front-End ist sowohl in der Lage eine synchrone Kommunikation mittels HTTP als auch eine asynchrone Kommunikation mittels 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 der 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 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 den 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 bestimmten Format an das Backend / den Server - die Nachrichten-Inhalte werden mittels Jackson im JSON-Format serialisiert und erzeugt, und auf der jeweiligen Endseite deserialisiert (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 Frontends.

### Login-Screen

Der Login-Screen enthält vier Schaltflächen für die Interaktion des Benutzers. Zwei Texteingaben zur Angabe eines Benutzernamen und eines Passworts – und zwei Schaltflächen / Buttons um die Benutzerangaben (Name, Passwort) mittels Login oder Registrierung an das Backend zu übertragen. Beide Prozesse werden im Folgenden näher beschrieben.

### Home-Screen

Der Home-Screen stellt die zentrale Benutzeroberfläche der Anwendung dar. Der Benutzer erhält darauf Zugriff nach einem erfolgreichen Anmeldeversuch. Der Home-Screen bietet die Möglichkeit, alle aktiven Spieler und alle aktiven Spiele des Systems 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.

**Chat Schaltfläche** – Ermöglicht den Anwendern den Austausch von Textnachrichten in Echtzeit.

**Aktive Spieler** - Zeigt alle aktiven Spieler (am System angemeldete Benutzer) tabellarisch an.

**Aktive Spiele** - Zeigt alle aktiven Spiele (Spielrunden des Systems) tabellarisch an. Ein Spieler kann an einem Spiel teilnehmen (sofern das Spiel noch nicht begonnen wurde oder ein Spieler-Slot verfügbar ist).

### Spielhistorie-Screen

Die Benutzeroberfläche für die Spielhistorie zeigt dem Benutzer alle in der Vergangenheit gespielten Spiele an welche im Backend persistiert worden.

### 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 sich in einem angemeldeten Spiel eintragen. Der Anwender welcher das Spiel definiert hat gilt als Admin und kann dadurch bestimmen wann das Spiel gestartet wird.

## Snake-Implementierung

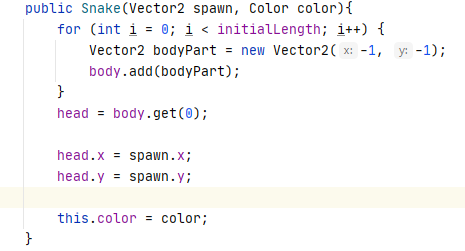
Für das Spiel wird ein Spielfeld anhand verschiedener Parameter generiert, wobei über *rows* und *columns* ein zwei dimensionales Koordinaten-System aufgespannt wird. Mit diesem Koordinaten-System sowie einem Timer, lassen sich alle jeweiligen Operationen realisieren. Das Spielfeld wird dabei auf einer *Stage* aufgebaut, dass diese losgekoppelt von dem laufenden Menü berechnet werden kann.

Aufbauend auf diesem Konzept betrachten wir nun Kernelemente des Spiels Snake:

### Schlangenexistenz und -bewegung

Schlangeninitialisierung:

Es wird auf einer vorgegebenen Koordinate ein Punkt erzeugt, welcher mit dem nächsten Tick die x und y Koordinate um 1 dekrementiert und an selber erneut einen Punkt erzeugt, bis die Anzahl an Punkten (Startschlangenlänge) erreicht ist. (Alternative Lösung: Es kann eine Liste erzeugt werden, an Koordinate x/y mit n Einträgen (n = Angegebene Initiallänge) an einer Koordinate Koordinate und einem Default Richtungsvektor für den Schlangenkopf. (Schlangenkopf = Schlangeliste(0)).



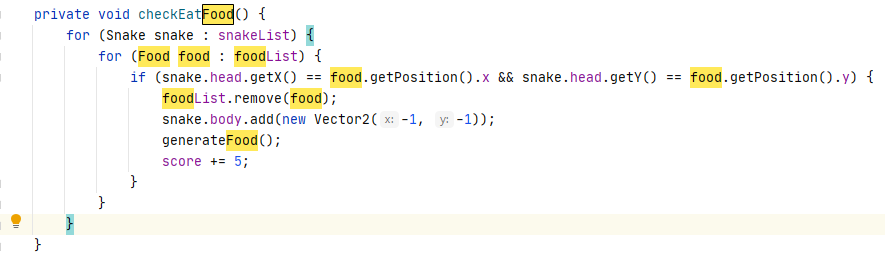
#### Schlangenbewegung

Die Bewegung der Schlange funktioniert, indem jedes Listenelement der Schlangenliste sich das Vorelement der Liste als neue Position holt, das letzte Listenelement entfernt wird und der Kopf der Schlange (snake.head, definiert als snake.body(0)) auf das Feld, errechnet aus snake.head, snake.currentDirection und der Eingabe (Eingabe realisiert über KeyFrames), gesetzt wird.



### Power-Ups/Food Generierung und Essen

Power-Ups/Food wird über eine Liste aus einer Koordinate und einem Bild realisiert. Dabei wird auf einer zufälligen Stelle des Koordinatensystems ein Essensteil generiert. Ist ein Schlangenkopf auf derselben Koordinatenposition hat dies die Auswirkung (hier), dass der Schlange ein weiteres Listenelement angehängt wird und generateFood() aufgerufen wird.



Die Realisierung des Essens als Liste ist, um mehrere Essenselemente gleichzeitig generieren zu lassen. (Regeln hierfür/sowie Power-Ups werden durch Zufallsvariablen organisiert)

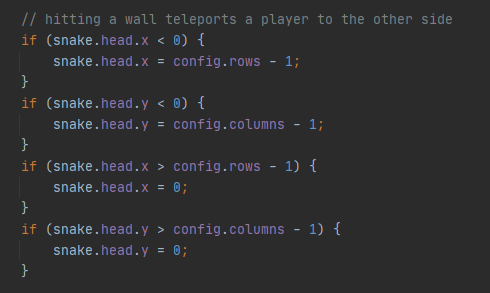
### Aktionen (Schlange, trifft Wand, andere Schlange, Essen, sich selbst…)

Eine jede Aktion lässt sich dann gleich der Essensaufnahme über einen einfachen Check, ob ein Schlangenkopf sich auf bestimmten Koordinaten befindet implementieren. Je nach Element, können verschiedene Regeln in Kraft treten.

Beispiel: Schlange GameOver bei Wand Kollision oder sich selber beißen:



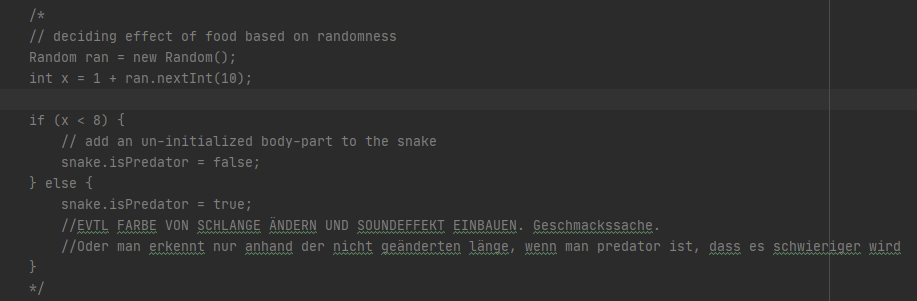
Beispiel: Alternative Spielregel: Wand-Teleport:



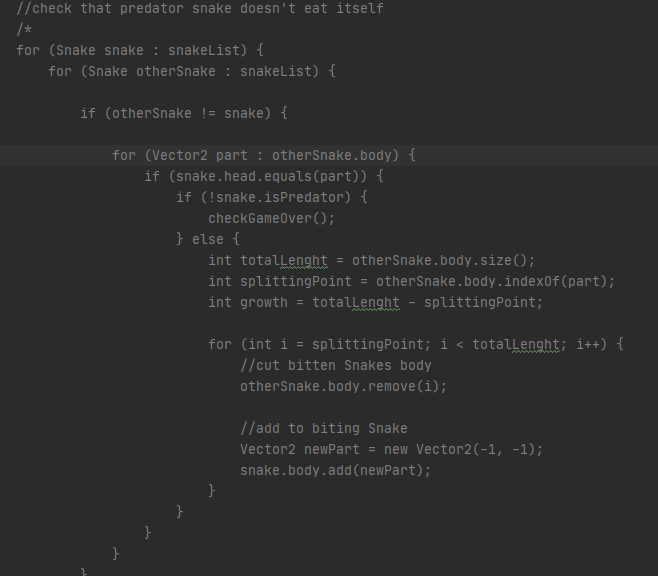
Beispiel: Schlange beißt einer anderen was ab:

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 einfach die Koordinate der beißenden Schlange x mal in die Liste eingefügt wird (ähnlich der Initialisierungsmethodik). Dass dies durch ein Power-Up realisiert werden kann, haben Schlangen das Attribut isPredator, was default false ist und durch ein solches PowerUp auf True gesetzt werden kann. Ist es auf True wird diese Logik aktiviert und wird mit dem Essen eines anderen Power Ups wieder auf false gesetzt.

Auswirkung je nach Essen:



Schlange beißt anderer was ab:



# 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 **persistance** 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 – welche sowohl als In-Memory als auch als File-Storage variante verwendet werden kann. Dieses Vorgehen erweist sich insbesondere für das Aufsetzen der Anwendung als Hilfreich, da die Installation und Konfiguration des RDBMS entfällt. Des Weiteren vereinfacht das Vorgehen die Realisierung und Nutzung von Unit-Tests.

Für die Realisierung mittels der Programmiersprache Java wird das in Spring enthaltene Spring-Data JPA verwendete, welches eine Spezifikation für JPA (Java Persistence Layer) darstellt. Hibernate wird als Implementierung für das ORM-Framework verwendet und direkt von Spring-Data genutzt.

Daten welche im RDBMS persistiert werden sind bspw. die Spielhistorie, welche das Datum des jeweiligen Spiels, die Spiel-Teilnehmer sowie deren Highscores enthält.

### H2 Database Engine

Ein weiterer Vorteil der Nutzung von H2 ist das in der Abhängigkeit enthaltenen Weboberfläche zur Verwaltung der Datenbank, sodass auch die Installation und Konfiguration eines solchen Tools entfällt:



Abbildung – Login des User-Interfaces zur Verwaltung der In-Memory / File-Storage Datenbank.  
Die URL zur Datenbank lautet: jdbc:H2:mem:testdb.



Abbildung - User-Interface zur Verwaltung der Tabellen

## 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.

Die Schnittstelle bietet folgende Endpoints für die Kommunikation 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:

@MessageMapping("/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 wiederrum 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, die Teilnahme eines anderen Clients am jeweiligen Spiel sowie den Spielstart dar.

*Spielbetritt*

*Aufbau WebSocket-Verbindung*

Client 1

Login

Login

Login

Login

*Versand Login-Daten*

*Versand Statuscode*

*Versand Login-Daten*

*Versand Statuscode*

Wechsel Homescreen

Wechsel Homescreen

*Versand Spieldefinition, Client 1 ist Admin*

*Aktualisierte Lobby-Daten*

Definition neues Spiel

Erstelle neues Lobby-Spiel

Füge Benutzer zur Lobby hinzu

Füge Benutzer zur Lobby hinzu

Asynchron: Empfange Lobby-Daten

Aktualisiere Lobby-View

Aktualisiere Lobby-View

Selektiere Spiel   
in Lobby

Trete Spiel bei

Empfange PlayerJoinsGameMessage

*Aufbau WebSocket-Verbindung*

Bestätige WebSocket Anfrage

Bestätige WebSocket Anfrage

Asynchron: Empfange Lobby-Daten

Aktualisiere Lobby-View

Repliziere  
Lobby-Daten

*Versand Lobby-Daten*

Repliziere  
Lobby-Daten

*Versand Lobby-Daten*

*Versand – Initiale GameInputMessage*

*Bestätige Spielbeginn*

Start Spiel

Empfange GameInputMessage

Asynchron:   
Spielbeginn

Repliziere  
Spiel-Beginn

Wechsel zu  
Game-Stage

Wechsel zu  
Game-Stage

Client 2

Backend

Synchron - HTTP

Asynchron - WebSocket

# 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 das der Spieler die „Login“ Schaltfläche betätigen muss. Nach Empfang der Login-Daten wird im Backend innerhalb der ApiController.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 ApiController.getHistorie() aufgerufen welche wiederrum 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 wiederrum 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

Composite, Factory und Observer

// TODO

# 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.

## Spieler-Teleport

TODO: Worte zum teleport von Spielern.

# 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

# Projektplan

|  |  |  |
| --- | --- | --- |
| Aufgabe | Datum | Autor |
|  |  | Benjamin Wulfert |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Verwendete Software

IntelliJ IDEA  
<https://www.jetbrains.com/de-de/idea/>

Git  
<https://git-scm.com/>

Apache Maven  
<https://maven.apache.org/>

H2 Database Engine  
<https://www.h2database.com/>

Spring Boot  
<https://spring.io/projects/spring-boot>

Spring Data JPA  
<https://spring.io/projects/spring-data-jpa>

Spring Messaging  
<https://spring.io/guides/gs/messaging-stomp-websocket/>

JavaFX  
<https://openjfx.io/>

Unirest  
<http://kong.github.io/unirest-java/>

SFXR  
<https://www.drpetter.se/project_sfxr.html>