Project - Memory Spiel

Stefan Klöss-Schuster

Vanessa Krohn

Ilias Sulgin

Alexander Ruhl

Daria Beck

Table of Contents

Setup

Bevor die Applikation verwendet werden kann müssen folgende Schritte durchgeführt werden:

1. Verwendet wird das BaseX plattform-unabhängige ZIP-Package. Der Ordner *memory*, der den Source Code enthält, muss in dem Ordner *basex/webapp* abgelegt werden.

2. BaseX kann von Haus aus nur mit XSLT Version 1.0 umgehen. In dieser Applikation werden XSLT Version 2.0 Stylesheets verwendet, um eigene XSLT-Funktionen definieren zu können. Hierfür muss ein anderen XSLT-Prozessor, z.B. der Saxon XSLT Processor [http://www.saxonica.com/], in den Klassenpfad eingebunden werden. Dazu muss nur die entsprechende .jar Datei in den Ordner *lib/custom/* abgelegt werden. BaseX erkennt den neuen Prozessor automatisch.

3. Die Datenbank muss über die REST-Route */setup/createDB* erstellt werden. Wenn die Datenbank erfolgreich angelegt wurde erfolgt eine Weiterleitung auf die Startseite.

4. Die SVG-Karten Definitionen müssen über die REST-Route */setup/initSvg* erstellt werden. Die Karten-Templates sind in dem Stylesheet *svgTemplates.xsl* definiert und mit den Stylesheet Variablen aus *xsltParameters.xsl* parametrisiert. Die REST-Route führt eine XSLT-Transformation durch und speichert das Ergebnis unter *static/svgs/svgGameElements.svg*.

5. Da XSLT-Forms [http://www.agencexml.com/xsltforms] verwendet wird, muss auch dieses in den Ordner *basex/webapp/static* eingebunden werden

Die Applikation wurde mit dem Firefox 62.0 (64-Bit) getestet. Chrome 69.0.3497.100 (64-Bit) hat Probleme im Zusammenspiel mit XForms und XSLT-Forms. XForms <xf:action> Elemente werden

nicht richtig ausgeführt wodurch auf der LoadGame-Seite keine gespeicherten Spiele gestartet werden können. Außerdem funktioniert deshalb das Umschalten zwischen den Karten-Konfigurationen auf der Highscore-Seite nicht.

Use Cases

Memory Spiel kann in folgende Anwendungsfälle aufgeteilt werden:

- Spieler kann über die Startseite die Lounge anzeigen
- Spieler kann in der Lounge zwischen "New Game", "Load Game" und "Highscore" navigieren
- Spieler kann ein neues Spiel beginnen. Dabei besteht die Möglichkeit die Anzahl der Karten, die Anzahl der Spieler und die Spielernamen auszuwählen
- Spieler kann ein gespeichertes Spiel laden
- Spieler kann ein laufendes Spiel über eine Route fortsetzen (/game/"gameID")
- Spieler kann ein laufendes Spiel über das Spiel-Menü speichern
- Spieler kann ein laufendes Spiel über das Spiel-Menü beenden
- Spieler kann einen Highscore erreichen, wenn das abgeschlossen wurde
- Spieler kann Highscores für verschiedene Spielkonfigurationen betrachten

User Experience - Menüführung (UX)

1. Zu Beginn werden die SpielerInnen von einer Startseite begrüßt über welche sie durch den Play-Button zum Menü gelangen. Dort steht es den SpielerInnen frei, ein neues Spiel zu starten (Button: New Game), ein bereits angefangenes Spiel zu laden (Button: Load Game) oder zur Highscore-Liste zu gelangen (Button: Highscore).

2. Wenn die SpielerInnen auf die Option "New Game" klicken, werden sie aufgefordert in das Username-Feld ihren individuellen Nutzernamen einzugeben. Weiterhin müssen sie entscheiden, mit welcher Anzahl von Karten (12,20,28) sie spielen möchten und wie viele TeilnehmerInnen es geben soll. Nachdem alles Nötige angegeben wurde, kann das Spiel über den "Start Game"-Button gestartet werden. Auf eine echte Registrierung der Spieler wird zugunstigen eines "arcade-artigen" Spielerlebnis verzichtet.

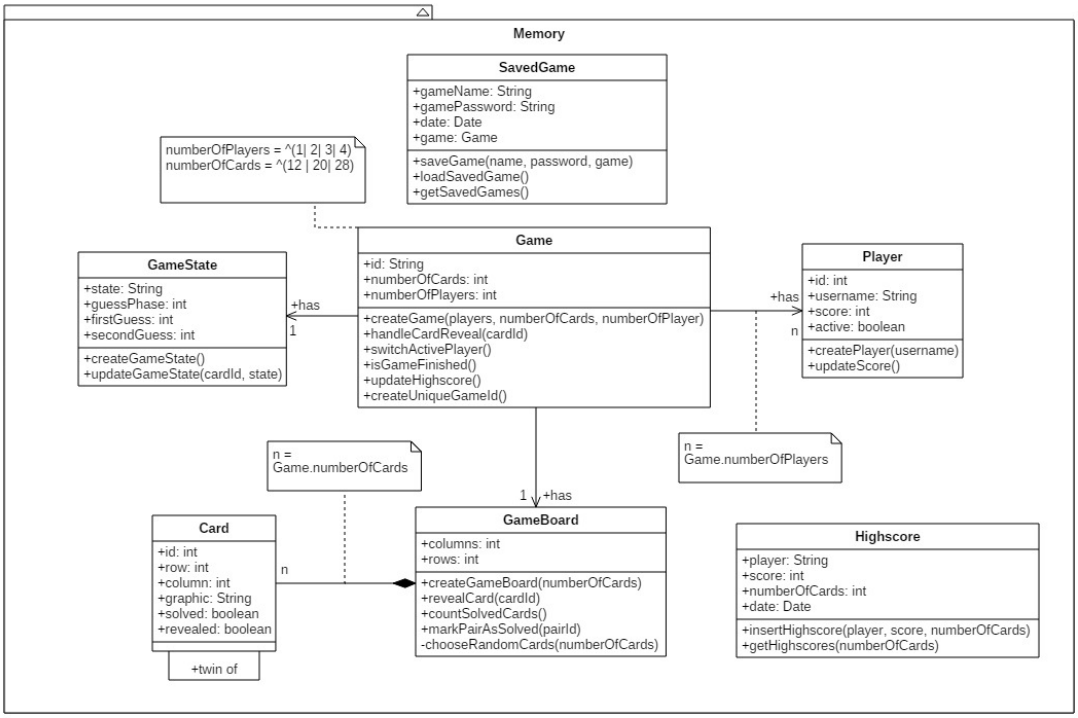
3. Alternativ kann über Load-Game im Menü auch ein bereits gespeichertes Spiel fortgesetzt werden. Die SpielerInnen wählen einfach ihr gespeichertes Spiel aus und setzen dieses mit Klick auf den "Start"-Button fort. Daraufhin erscheint ein Dialog in dem die SpielerInnen das Passwort für das gespeicherte Spiel eingeben müssen. Bei Klick auf den "Cancel-Button" wird der Dialog geschlossen. Mit einem Klick auf den "Load Game"-Button wird das Spiel geladen, falls das Passwort stimmt. Falls nicht werden die SpielerInnen zum Menü weitergeleitet.

4. Sobald ein Spiel gestartet wird, erscheinen umgedrehte Karten (je nach Auswahl der Kartenanzahl) in der Mitte vom Spielfeld. Am Rand sind die TelnehmerInnen und deren jeweilige Punktzahl gelistet. Die Spielregeln sind die des altbekannten Memorie-Spiels. Die SpielerInnen können jederzeit ihr Spiel abspeichern. Sobald ein Spiel beendet wurde, wird die Spielerin oder der Spieler mit den meisten Punkten als "Gewinner" markiert und die besten Spieler erhalten einen Eintrag in der Highscore-Liste.

5. Die Highscore-Liste ist ebenfalls über das Menü erreichbar und besteht aus den Nutzernamen und Punktzahl der Zwanzig Spielerinnen und Spieler.

UML-Klassendiagramm

Die benötigten Zustandsdaten für ein laufendes Spiel und die Hilfsdaten (Highscore, SavedGame) sind im UML-Klassendiagramm



dargestellt.

Datenbeschreibung

Die folgende Liste beschreibt die Klassen des oben dargestellten UML-Klassendiagramm, die für ein laufendes Spiel benötigt werden:

- *Game*: Die Spiel-Klasse repräsentiert eine Instanz eines laufenden Spiels. Das Attribut "id" ist über alle laufenden und gespeicherten Spiele hinweg eindeutig und identifiziert ein Spiel. Ein Spiel wird mit 12, 20 oder 28 Karten gespielt ("numberOfCards") und hat bis zu vier Spieler ("numberOfPlayers"). Über die Spiel-Klasse kann auf die anderen Klassen, welche für ein laufendes Spiel benötigt werden, zugegriffen werden. In XML-codiert ist damit "Game" der Container für die anderen Zustandsklassen.

- *GameState*: Diese Klasse repräsentiert Informationen, die für den Spielablauf benötigt werden. Das Attribute "state" vom Typ String gibt an, ob das Spiel beendet ist ("finished") oder noch läuft ("running"). Das Feld "guessPhase" gibt an in welcher Ablaufphase sich das Spiel momentan befindet. Zum Beispiel repäsentiert die "guessPhase=0" die Phase in der die SpielerInnen die ersten Karte aufdecken. Die Karten-Id des ersten und des zweiten Tipps werden in den Feldern "firstGuess" bzw. "secondGuess" gespeichert.

- *Player*: Die Spieler-Klasse repräsentiert einen von maximal vier Spielern. Sie hat eine "id" und einen "username", um einen Spieler zu identifizieren. Jeder Spieler hat einen Integer "score", der die aktuelle Punktzahl des Spielers festhält. Das Attribut "active" gibt an, ob der Spieler gerade an der Reihe ist.

- *GameBoard*: Das Spielfeld besteht aus den Karten die aktuell im Spiel sind. Die Attribute "columns" und "rows" bestimmen die Ausmaße des Spielfelds.

- *Card*: Karten werden in Paaren dargestellt. Jede Karte erhält eine eindeutige Id und eine pairId (twin of Assoziation) welche für Zwillingkarten identisch ist. Die Klasse enthält die boolean Attribute

"revealed" und "solved". Sie geben an, ob die Karte aufgedeckt ist und ob die Zwillingskarte bereits gefunden wurde. Der String "graphic" assoziiert die Karte mit einem Bilderdatei, die dargestellt wird, wenn die Karte aufgedeckt ist. Die Felder "row" und "column" bestimmen die Position der Karte auf dem Spielbrett.

Die folgende Aufzählung beschreibt Klassen in denen Hilfsdaten gespeichert werden:

- *Highscore*: Die Klasse Highscore wird benötigt, um die Highscores der Spieler zu speichern. Eine Instanz der Klasse wird nur gespeichert, wenn das Spiel beendet wurde und ein Spieler einen Highscore erreicht hat. Mit Hilfe das Attributs "numberOfCards" kann zwischen den Highscores verschiedener Spielkonfigurationen unterschieden werden. Das Attribut "player" gibt den Namen des Spielers an, "score" enthält die erreichte Punktzahl und "date" hält da fest, wann der Highscore erreicht wurde.

- *SavedGame*: Die Klasse repäsentiert ein gespeichertes Spiel. Neben den Zustandsdaten des gespeicherten Spiels, hat die Klasse die Attribute "gameName" und "gamePassword" welche zum Laden des Spiels benötigt werden. Zusätzlich wird in dem Feld "date" festgehalten, wann das Spiel gespeichert wurde.

Die Attribut-Typen, die in dem UML-Klassendiagramm angegeben wurden, dienen lediglich zum besseren semantischen Verständnis. Bei der Implementierung kann es vorkommen, dass ein Integer (xs:int) als String (xs:string) repräsentiert wird. Außerdem sind die Zustandsdaten nicht minimal d.h. Informationen werden teilweise redundant gespeichert. Zum Beispiel kann der Wert des Attributs "Game.numberOfCards" über die Assoziation der GameBoard-Klasse mit der Karten-Klasse bestimmt werden. Der Vorteil des expliziten Speicherns wichtiger Spiel-Parameter ist das einfache Auslesen dieser Werte.

Methodenbeschreibung

In diesem Kapitel werden einige Methoden aus dem obigen Klassendiagramm erläutert.

- Game.createGame(...): Erstellt den Spielzustand. Ruft Konstruktoren der anderen Klassen auf wie z.B. GameState.createGameState(). Verwendet createUniqueGameId() um eine einzigartige Id aus dem Hash der Spielernamen zu erzeugen. Gibt den initialen Spielzustand zurück.

- Game.handleCardReveal(): Verändert den Spielzustand abhängig von der aufgedeckten Karte und dem aktuellen GameState.

- Game.isGameFinished(): Überprüft, ob das aktuelle Spiel zu Ende ist. Nutzt GameBoard.countSolvedPairs() und numberOfCards.

- Game.switchPlayers(): Setzt einen anderen Spieler aktiv. Wird verwendet wenn ein Kartenpaar nicht gefunden wurde

- Game.updateHighscore(): Veranlasst eine Aktualisierung des Highscores, wenn das Spiel beendet ist. Ruft Highscore.insertHighscore() auf.

Architektur

Überblick

Die Architektur der Anwendung entspricht der in der Vorlesung vorgestellten Model-View-Controller-Architektur mit einer passiven View.

Die Views werden mit HTML5, SVG und XForms im Zusammenspiel mit XSLTForms realisiert und in einem Browser dargestellt. Die eigentliche Spiel-Seite wird aus dem Spielzustand mit Hilfe eines XSLT-Programms generiert. Ausgabeformat ist hierbei HTML5 mit eingebettetem SVG. Die Views

bietet den SpielerInnen Interaktionsmöglichkeiten über Links (<a/>), HTML Dialoge (<dialog/>) und HTML Forms. Wenn die SpielerInnen eine Interaktion vornehmen, wird ein HTTP-Request mit einer vordefinierten Route abgesendet und die Antwort erneut im Browser dargestellt.

Die Controller-Module haben die Aufgabe die von den Views entsendeten HTTP-Requests zu verarbeiten. Das Mapping von HTTP-Requests auf XQuery Methoden erfolgt mit RESTXQ-Annotationen. Zur Verarbeitung der HTTP-Requests werden die Funktionen in den Model-Modulen benötigt. Ein Design-Ziel ist hierbei den Controller möglichst "dumm" zu halten um eine starke "separation of concerns" zu ermöglichen. Das Model hat aus diesem Grund eine REST-Schnittstelle, die wiederum über RESTXQ-Annotationen realisiert wird. Der Controller greift auf die Model-API über Wrapper-Methoden zu, welche mit http:send-request implementiert sind. Der Controller generiert aus den zurückgegebenen Informationen eine neue View, die in dem Browser dargestellt wird.

Das Model kann weiterhin in zwei Schichten unterteilt werden, die Logik und die Datenbank-Funktionen. Nur die Controller-Module dürfen die Model-Funktionen aufrufen, niemals die Views direkt. Grundsätzlich ist gewollt, dass nur über die Logik-Schicht auf die Datenbank-Funktionen zugegriffen wird aber es gibt Außnahmen, falls keine vorgeschaltete Logik benötigt wird. Ein Beispiel hierfür ist das Anlegen der Datenbank. Aufgabe der Logik-Schicht ist das Manipulieren der Daten.

Zugriff auf alle Datenbank-Funktionen, sowohl lesend als auch schreibend, erfolgt ebenfalls über eine eigene Datenbank-REST-API, die mit RESTXQ Annotationen umgesetzt wird. Ein Vorteil dieser Entscheidung ist, dass die aufrufende Methoden einfacher gestaltet werden kann. Datenbank-Updates und Lesezugriffe können an beliebiger Stelle in der Funktion ausgeführt und vermischt werden. Die Pending Update List wird für jede Aufruf einer Datenbank-Funktion direkt verarbeitet. Dadurch muss nicht auf die implizite Update-Reihenfolge geachtet werden. Außerdem werden jegliche Deadlocks verhindert und das Prinzip der "seperation of concerns" umgesetzt.

REST-API

Folgende REST-API wird verwendet, um die Applikation zu initialisieren:

Table 1. Setup REST-API

Controller Module	Controller Function	Method	Path	Beschreibung
setupController.xqm	sc:createDatabase	GET	/setup/createDB	Initialisiert die Datenbank.
	sc:dropDatabase	GET	/setup/dropDB	Löscht die Datenbank
	sc:initSvg	GET	/setup/initSvg	Erstellt eine Datei mit Karten SVG-Definitionen aus einem XSLT-Programm (svgTemplates.xsl), dass die Kartentemplates enthält. Die Kartentemplates werden mit Variablen in xsltParameters.xsl parametrisiert.

Folgende REST-API wird in der Controller-Schicht umgesetzt und stellt alle Interaktionsmöglichkeiten des Clients dar:

Table 2. Controller REST-API

Controller Module	Controller Function	Method	Path	Beschreibung
lobbyController.x-qm	lc:start	GET	/	Gibt die Start-Seite als HTML5 zurück
	lc:menu	GET	/menu	Gibt das Lobby Menü als HTML5 zurück
	lc:createGame	GET	/createGame	Gibt die CreateGame-Seite zurück. Diese Seite verwendet XForms und wird mit Hilfe von XSLTForms umgewandelt.
	lc:loadGame	GET	/loadGame	Gibt die LoadGame-Seite zurück. Diese Seite verwendet XForms.
	lc:getSavedGames	GET	/savedGames	Befüllt die XForms Instanz der LoadGame-Seite. Liefert alle gespeicherten Spiele zurück.
	lc:highscores	GET	/highscores	Gibt die Highscores-Seite zurück. Diese Seite verwendet XForms.
	lc:getHighscores	GET	/getHighscores	Befüllt die XForms Instanz der Highscores-Seite. Liefert alle Highscores der entsprechenden Karten-Konfiguration zurück. Konfiguration wird mit dem Query-Parameter numberOfCards angegeben.
gameController.x-qm	gc:createGame	POST	/game/create	Erstellt ein neues Spiel anhand der Parameter numberOfCards, numberOfPlayers und den Spielernamen im Body des Requests. Leitet danach auf die Spielseite weiter.

Controller Module	Controller Function	Method	Path	Beschreibung
	gc:createGamePage	GET	/game/{$gameId}	Gibt eine spezifische Spielseite zurück. Die Seite wird aus dem Spielzustands mit einem XSLT-Programm generiert.
	gc:revealCard	GET	/game/{$gameId}/revealCard/{$cardId}	Diese Route wird aufgerufen wenn ein Spieler eine Karte auswählt hat und verändert den Spielzustand entsprechend. Ersetzt den alten Spielzustand in der Datenbank. Überprüft, ob das Spiel zu Ende ist, speichert eventuell den Highscore und leitet entweder zur Spielseite oder zum Hauptmenü zurück.
	gc:loadGame	POST	/game/load	Wird von der LoadGame-Page augerufen. Body enthält das vom User eingegeben Passwort und die GameId. Überprüft, ob das Passwort übereinstimmt. Falls ja, aktiviert das gespeicherte Spiel und leitet zu der Spiel-Seite weiter. Ansonsten wird zum Lobby-Menü weitergeleitet.
gameMenuController.xqm	gmc:saveGame	POST	/gameMenu/saveGame	Wird über das InGame-Menü aufgerufen. Speichert das aktuelle Spiel als savedGame in der Datenbank
	gmc:ExitGame	POST	/gameMenu/exit	Löscht das Spiel ein leitet zur LobbyMenu-Seite weiter. Die gameId wird über eine

Controller Module	Controller Function	Method	Path	Beschreibung
				HTML-Form übermittelt.

Mit folgender REST-API kann die Logik-Schicht des Models verwendet werden

Table 3. Model Logic REST-API

Model Module	Controller Function	Method	Path	Beschreibung
gameBoardFunctions.xqm	gbf:handleCardReveal	POST	/model/game/{$gameId}/revealCard/{$cardId}	Verändert den Spielzustand eines besimmten Spiels anhand des gespeicherten Spiel-Zustands und der aufzudeckenden Karte. Gibt den veränderten Spiel-Zustand zurück
gameConstructor.xqm	gco:createGame	POST	/model/game/create	Erstellt den Spielzustand anhand der XML-codierten Parameter im Body des Requests. Parameter sind die Anzahl der Karten, die Anzahl der Spieler und die Spielernamen. Gibt den XML-codierten Spielzustand zurück.
gameMenuFunctions.xqm	gmf:saveGame	POST	/model/gameMenu/saveGame	Erstellt das savedGame anhand der XML-codierten Parameter im Body des Requests. Parameter sind der Spielname, das Passwort und die GameId. Erstellt eine neue einzigartige GameId und speichert das savedGame in der Datenbank
highscoreFunctions.xqm	hf:saveHighscores	POST	/model/highscore/save	Erstellt Highscores. Der Body des Requests erhält die Spielernamen und die Scores

Model Module	Controller Function	Method	Path	Beschreibung
				eines beendeten Spiels. Die globale Variable hf:highscoreLimit bestimmt wieviele Highscores für eine Kartenkonfiguration gespeichert werden. Die alten Highscores der Konfiguration werden aus der Datenbank gelöscht und die neuen Highscores in die Datenbank eingefügt.

Mit folgender REST-API kann lesend und schreibend auf die Datenbank zugegriffen werden:

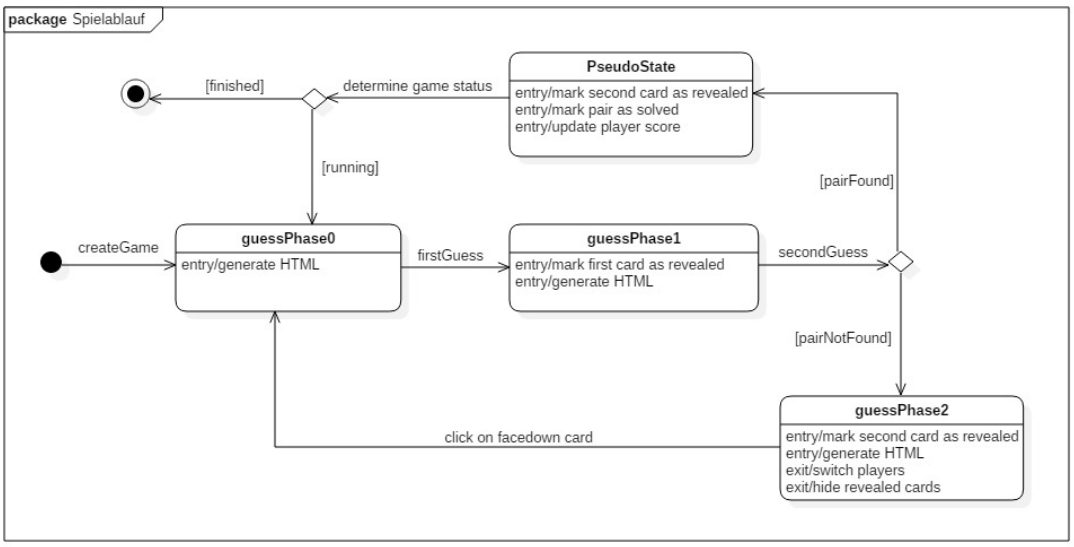
Table 4. Model Database REST-API

Model Module	Controller Function	Method	Path	Beschreibung
databaseFunctions.xqm	%updating dbf:initDatabase	POST	/model/database/init	Erstellt eine Datenbank mit Container-Nodes für die laufenden Spiele, die gespeicherten Spiele und die Highscores. Wird nur erstellt wenn die Datenbank noch nicht existiert.
	%updating dbf:dropDatabase	POST	/model/database/drop	Löscht die Datenbank
	dbf:getGame	GET	/model/database/getGame/{$id}	Gibt den Spielzustand mit der entsprechenden gameId zurück
	%updating dbf:createGame	POST	/model/database/createGame	Speichert den Spielzustand im Body des Requests in der Datenbank
	%updating dbf:replaceGame	POST	/model/database/replaceGame	Ersetzt einen alten Spielzustand mit einem Neuem. Der neue Spielzustand ist im Body des Requests
	%updating dbf:deleteGame	POST	/model/database/	Löscht den Spielzustand mit

Model Module	Controller Function	Method	Path	Beschreibung
			base/deleteGame/{$id}	der übergebenen gameId aus der Datenbank
	%updating dbf:createSaveGame	POST	/model/database/createSaveGame	Fügt ein savedGame aus dem Body des Requests in die Datenbank ein
	dbf:getAllSavedGames	GET	/model/database/getAllSavedGames	Liefert alles savedGames, die in der Datenbank gespeichert sind, zurück
	dbf:getSavedGame	GET	/model/database/getSavedGame/{$id}	Gibt ein savedGame mit der entsprechenden gameId aus der Datenbank zurück
	%updating dbf:deleteSavedGame	POST	/model/database/deleteSavedGame/{$id}	Löscht ein savedGame mit der entsprechenden gameId aus der Datenbank
	dbf:getHighScore	GET	/model/database/highscores/{$numberOfCards}	Gibt alle Highscores mit der entsprechenden Kartenkonfiguration aus der Datenbank zurück
	%updating dbf:deleteHighScores	POST	/model/database/deleteHighscores/{$numberOfCards}	Löscht alle Highscores mit der entsprechenden Kartenkonfiguration aus der Datenbank
	%updating dbf:insertHighScore	POST	/model/database/insertHighscores	Fügt alle Highscores, die im Body des Requests sind, in die Datenbank ein
	dbf:gameIdExists	GET	/model/database/gameIdExists/{$id}	Gibt zurück, ob die übergebene gameId bereits in einem laufendem oder gespeicherten Spiel existiert.

Spielablauf

Folgendes UML Zustandsdiagramm zeigt das Konzept des Spielablaufs:



Zunächst wird das Spiel über die Route */game/create* angelegt. Alle relevanten Informationen zum Erstellen des Spiels sind im Body des Requests enthalten:

```
<data> <numberOfCards>20</numberOfCards> <numberOfPlayers>1</num-
berOfPlayers> <players>John</players> </data>
```

Im obigen Beispiel wird ein Spiel mit einem Spieler namens John und 20 Karten erstellt. Das Model Modul *gameConstructor* wird von dem *gameController* über die REST-Route */model/game/create* angesprochen und gibt die XML-codierten Daten im Body des Requests weiter. Die Model Funktion generiert aus den Spielinformationen einen fertigen Spielzustand. Dabei wird eine einzigartige gameId gewählt und zufällige Kartenmotive, die in *static/svgs/cardSet.xml* festgehalten sind, ausgewählt. Der Spielzustand wird über die HTTP-Response and den gameController weitergeleitet.

Der gameController ruft die Route */model/database/createGame* auf, welche den Spielzustand in der Datenbank speichert. Dann erfolgt eine Weiterleitung zur eigentlichen Spiel-Seite */game/{$gameId}*. Aus unbekannten Grund ändert die Weiterleitung mit *web:redirect* hier nicht die URL, die der Browser anzeigt.

Die gameController Route*/game/{$gameId}* hat die Aufgabe aus dem Spielzustand eine entsprechende View zu erzeugen. Hierfür wird das XSLT-Programm *src/model/xslt/stateToHTML.xsl*, welches nach dem Pull-Prinzip arbeitet, verwendet. Das Ergebnis-Skelett ist bereits vorgegeben und die benötigten Informationen werden aus dem XML-codierten Spielzustand extrahiert. Zur Darstellung des Spiels und Berechnen des Layouts werden Hilfsfunktionen aus dem Stylesheet *src/model/xslt/xsltTemplates.xsl* verwendet. Das Stylesheet enthält außerdem Variablen zum Parametrisieren der Karten-SVGs und der Spieler-Informationen. Die Karten-SVGs müssen bei dem Setup der Applikation über die Route */setup/initSvg* zunächst initialisiert werden. Die Route stößt das XSLT-Programm *src/model/xslt/svgTemplates.xsl* an und schreibt die fertigen Karten-Definitionen in die Datei *static/svgs/svgGameElements.svg*. Der Vorteil dabei ist, dass alle Karten-Grafiken vollständig parametrisiert sind und schnelle Änderungen am Design möglich werden.

Jede verdeckte Karte wird in dem XSLT-Programm *src/model/xslt/stateToHTML.xsl* mit einem Link zur Route */game/{$gameId}/ revealCard/{$cardId}* versehen. Die gameId und cardId wird dabei automatisch richtig parametrisiert. Wenn die Spieler einen Karte auswählen wird diese Route aufgerufen, um die entsprechenden Änderungen am Spielzustand vorzunehmen.

Der Spielablauf der im obigen UML-Zustandsdiagramm zusammengefasst ist, folgt immer dem selben Zyklus:

1. Spieler wählt eine Karte aus und die Route */game/{$gameId}/ revealCard/{$cardId}* wird aufgerufen

2. Die entsprechende gameController-Funktion ruft die Model-Route */model/game/{$gameId}/ revealCard/{$cardId}* auf, um den Spielzustand zu manipulieren.

3. Der Spielzustand wird je nach *guessPhase* in der sich das Spiel aktuell befindet angepasst und an den gameController zurückgegeben. Die guessPhase wird im Spielzustand unter *game/gameState/guessPhase* gespeichert.

   - Die *guessPhase = 0* repräsentiert den Zustand in dem der Spieler noch keinen Tipp abgegeben hat.

   - Die *guessPhase = 1* repäsentiert den Zustand nach dem der Spieler die erste Karte aufgedeckt hat.

   - Beim Aufdecken der zweiten Karte erfolgt eine Fallunterscheidung: Falls die zwei Karten nicht identisch sind wird das Spiel in den Zustand *guessPhase = 2* versetzt. Falls die Karten identisch sind wird eine Reihe von Anpassungen am Spielzustand vorgenommen und das Spiel wieder in den Zustand *guessPhase = 0* versetzt: Das Kartenpaar wird als "solved" markiert, der Spieler-Score wird aktualisiert und es wird überprüft, ob alle Kartenpaare gefunden wurde. Wenn alle Kartenpaare gefunden sind, wird "finished" in *game/gameState/state* hinterlegt. Der Controller überprüft dieses Flag später.

   - Die *guessPhase = 2* ist der Zustand der es ermöglicht, dass die fälschlicherweise aufgedeckten Karten zunächst offen liegen bleiben, damit die Spieler sich die Karten einprägen können. Bei Klick auf eine verdeckte Karte werden die fälschlicherweise aufgedeckten Karten wieder umgedreht, der aktive Spieler gewechselt und in die guessPhase = 0 zurückgekehrt. Eine Überprüfung, ob das Spiel beendet ist, erübrigt sich, da das Spiel nur nach dem Auffinden von zwei identische Karten beendet werden kann.

4. Der gameController ersetzt den alten Spielzustand mit dem neuen Spielzustand über die Route */model/database/replaceGame*.

5. Der gameController überprüft, ob das Spiel fertig ist anhand des Werts im Spielzustand *game/gameState/state*. Wenn das Spiel beendet ist wird die Highscore-Logik über die Route */model/highscore/save* angestoßen, um eventuelle Highscores der Spieler zu speichern. Das Spiel wird über die Route */model/database/deleteGame/{$gameId}* aus der Datenbank gelöscht und es erfolgt eine Weiterleitung auf die LobbyMenu-Seite. Wenn das Spiel noch nicht fertig ist erfolgt eine Weiterleitung über */game/{$gameId}*, um den aktualisierten Spielzustand zu einer View zu transformieren. Hier beginnt der Zyklus erneut.

Ein Problem bei dieser Umsetzung des Spiels ist, dass jedes Mal wenn eine Karte gewählt wurde ein HTTP-Request gesendet wird und der Client auf die Antwort warten muss. Außerdem wird dadurch die Spielseite ständig neu geladen. Eine andere Lösung wäre aber mit Logik im Client verbinden, zum Beispiel JavaScript oder XForms. Auf JavaScript wurde im ganzen Projekt verzichtet, um die Projektvorgaben einzuhalten. Auf XForms mit XSLTForms wurde auf der Spiel-Seite ebenfalls verzichtet, um ständige XSLT-Transformationen zu vermeiden und die Komplexität im Zaum zu halten.

Reflektion

Organisation der Arbeit

Zur Versionsverwaltung des Source Codes wurde Git im Zusammenspiel mit GitHub verwendet. Sämtliche Arbeitsprodukte sind in dem Repository https://github.com/kloessst/memory zu finden. Als Entwicklungsprozess wurde ein vereinfachtes GitFlow vorgeschlagen. Dabei darf der Master Branch immer nur funktionierende Product Increments enthalten. Neue Features werden ausschließlich in den Feature Branches entwickelt. Bei Fertigstellung eines Features wird ein Pull-Request erstellt und nach einem Review und Test durch einen zweiten Entwickler in den Master Branch gemerged. Leider kon-

nte dieser Prozess nicht vollständig umgesetzt werden, wegen Zeitmangel und unterschiedlichen Wissensständen im Umgang mit Git und GitHub.

Die Kommunikation erfolgte hauptsächlich über eine Whatsapp Gruppe und hat grundsätzlich sehr gut funktioniert. Während des Semesters wurden einige Treffen veranstaltet, um den Arbeitsablauf zu besprechen. Die restliche Arbeit am Projekt erfolgte remote. Die Verteilung der Aufgaben war problematisch. Es wurde versucht die Aufgaben nach Wissensständen zu verteilen. Da aber die Team-Mitglieder unterschiedlichste Hintergünde haben, wurde die Arbeitslast, vorallem bei der Implementierung, nicht fair verteilt. Das Verteilen der Arbeit nach Wissensstand hat außerdem zur Folge, dass die XStack-Applikation nicht von jedem vollständig überblickt wird. Zusätzlich wurde die Komplexität des Projekts unterschätzt und schließlich musste das Testat verschoben werden.

Praktikum

Die Aufgabe ein Memory Browser-Game zu entwickeln hat allen sehr gefallen. Bei einem Spiel besteht die Möglichkeit sich beim Design auszuleben. Außerdem sind Fortschritte schnell ersichtlich.

Der vorgegebene Technologie-Stack konnte den Programmier-Horizont deutlich erweitern. Das Arbeitet mit BaseX ging leicht von der Hand dank dem ausführlichen Wiki und der vielen vorgefertigten Module. Nachteil des XStacks ist vor allem das schwierige Debuggen von fehlerhaften Code, die oft veralteten Informationen im Web und die Tatsache, dass der XStack kaum Verwendung findet in der Wirtschaft.

Die Organisation Praktikums war von Anfang an klar verständlich. Die Tatsache, dass die Arbeitsblätter nicht abgegeben werden müssen, hilft den Studierenden, die eine hohe Arbeitsbelastung haben. Dies ermöglicht eine flexiblere Arbeitseinteilung. Jedoch besteht auch die Gefahr, dass der Anschluss schnell verpasst wird, da die XML-Technologien oft aufeinander aufbauen. Die Möglichkeit einen Notenbonus zu verdienen, wenn ein Arbeitsblatt präsentiert wird, ist eine gute Maßnahme dagegen.

Die Betreuung des Praktikums war sehr gut. Philipp Ulrich war über E-Mail und die Sprechstunden immer zu erreichen und hat schnelle und gute Antworten auf die Fragen gegeben. Prof. Dr. Anne Brüggemann-Klein war auch immer erreichbar. An dieser Stelle muss die freundliche und kulante Art von Ihr nochmals hervorgehoben werden.