
XForms Multiclient Blackjack Documentation

Daniel Meint <d.meint@tum.de>

Felix Hennerkes <ga38hom@mytum.de>

Janik Schnellbach <janik.schnellbach@tum.de>

Maximilian Karpfinger <maximilian.karpfinger@tum.de>

Table of Contents

| | |
|--------------------------------------------------------|----|
| Introduction | 1 |
| Installation | 2 |
| Description of the case study | 2 |
| Card Values | 2 |
| Course of a Round | 2 |
| Architecture | 4 |
| The component model | 5 |
| Modeling the model | 5 |
| Design decisions and discussion | 6 |
| Implementing the component Model | 7 |
| Implementation details, decisions and discussion | 11 |
| The component View | 12 |
| Modeling the component View | 12 |
| Implementing the component View | 14 |
| The component Controller | 14 |
| Modeling the component Controller | 14 |
| Implementing the component Controller | 14 |
| Collaboration | 15 |
| Conclusions | 16 |
| Bibliography | 16 |

Introduction

The so-called XStack allows us to build stable and highly performant multi-client web applications using only XML technologies. This paper documents the design and implementation of the popular casino game Blackjack as a browser-based online multiplayer game and thereby demonstrates the XStack's capabilities. The system was developed during the summer semester 2019 in the context of the practical course *XML Technology* at the Technical University of Munich and is built only with the technologies and concepts introduced in the course.

The app is expected to run reasonably well in any modern browser with SVG support, but we strongly recommend to use a current version of *Mozilla Firefox*. Furthermore, for server-side processing, a current version of the XML database system and XQuery processor *BaseX* with support for the STOMP protocol is needed. An installation guide can be found on Github [BaseX, BaseXStomp]. The installation manual for our blackjack application is part of this document.

Our application can be used by any number of users and handles multiple games played in simultaneously. A client can spectate a game or join as an actively participating player. Unlimited clients can be subscribed to a game as spectators, but only up to five can play on a single table, i.e. one instance of a game, at the same time.

The paper is structured as follows. The next section describes the rules and conceptually analyses the various phases of a game of blackjack. It is followed by a section on the high-level architecture of our system, which is based on the Model-View-Controller pattern. Subsequently, three sections explain the components of an MVC architecture both from a modeling and technical perspective. Finally, we dedicate one section to reflect on the development methodologies we employed and the organization of the practical course.

Installation

As already stated before, the application requires the STOMP-capable BaseXHTTP Server with a corresponding environment within the BaseX webapp folder. The folder `xforms-multiclient` needs to be placed directly inside `webapp`, the folder `xforms-static` goes inside `webapp/static`. One can then access `localhost:8984/xforms-multiclient/setup` using the Firefox web browser in order to initialize the required databases (`xforms-games` and `xforms-users`). Afterwards, calling `localhost:8984/xforms-multiclient` should open the game environment. Before logging in, a user might need to register a new account by clicking the '+' symbol on the right side.

Description of the case study

We follow a domain-driven design approach in the implementation of the web application. Therefore, we want to focus on the core domain logic of blackjack in this section. We explain the rules of the game and describe the order of events during a single round of blackjack.

Globally, there are a variety of rulesets with slight differences. We seek to present the most universally accepted variation, commonly found in Las Vegas casinos. Up to five *players* compete not against each other, but separately against the *dealer*. The objective of each player is to draw cards and maximize the sum of their respective values (*hand value*) without exceeding a sum of 21 (*bust*).

Card Values

The game is played with a single deck of French playing cards. Number cards are worth their value, e.g. the seven of hearts is worth seven points, face cards (Jack, Queen, and King) are worth ten points, and an Ace can be counted as either one or eleven, depending on what is more favorable in a specific situation. A hand containing an Ace counted as eleven is referred to as a *soft* hand because the value of the Ace will change to a one to prevent the player from busting if he was to draw another card and otherwise exceed 21. A card's suit is irrelevant in blackjack.

Course of a Round

Blackjack is round-based and each round consists of multiple stages. We distinguish between the betting, playing, and evaluation phase. After the end of a round, the next round can be initiated by any participating player. Players can leave the table at any time.

Betting

Before the actual playing begins, players place their bets. Our version uses US-dollars as currency and does not limit players in the amount they want to bet. It does, however, prohibit them from playing without betting at all. We will refer to this stage of the game as the *betting phase* throughout the following sections.

Playing

At the beginning of the *playing phase*, each player is dealt two cards face up. The dealer receives one exposed card that everyone can see and one hidden card.

The dealer subsequently asks each player, going clockwise around the table, whether they want to improve their respective hand by drawing additional cards. Each player has the following options:

Figure 1. Possible player decisions.

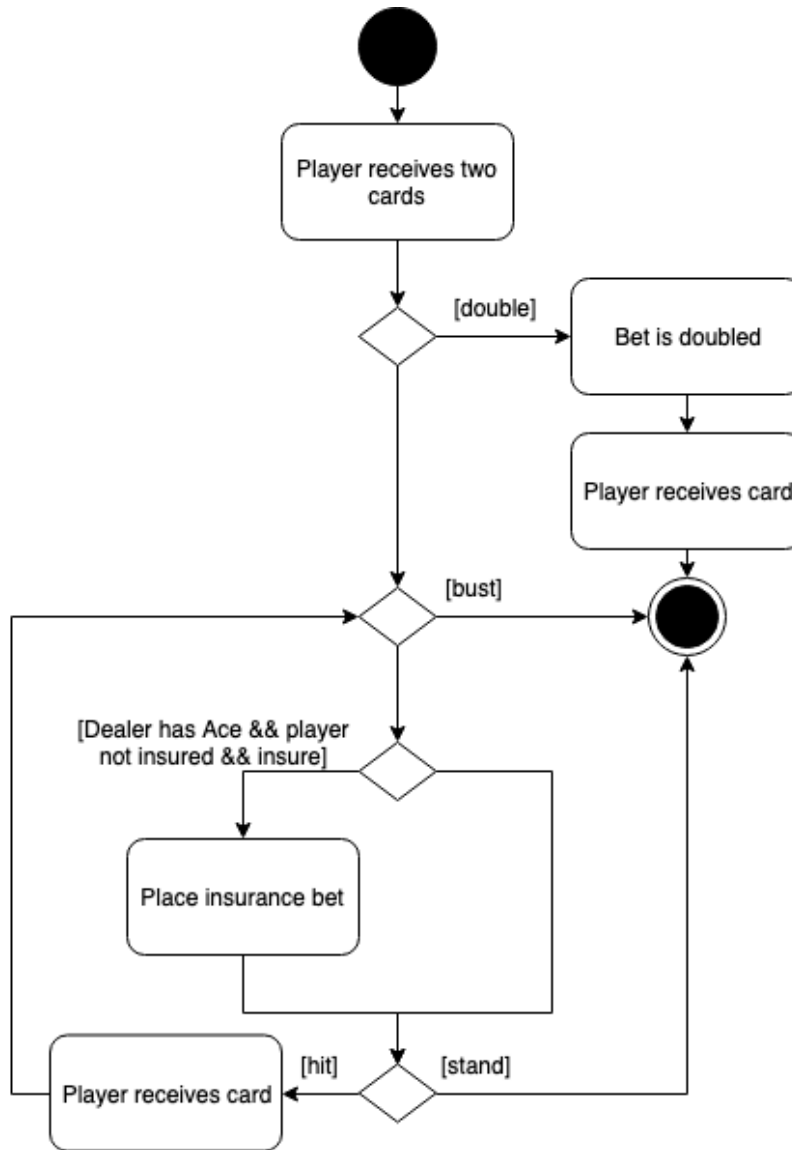


Table 1. Playing options explained.

| <i>Action</i> | <i>Description</i> | <i>Condition</i> |
|---------------|------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Stand | Ends the player's turn | Always possible and automatically performed when the player busts |
| Hit | The player receives another card from the deck | Available if the player's hand value is less than 21 |
| Double | The player doubles his bet and receives exactly one more card before finishing his turn | Available as the player's first action |
| Insurance | The player puts a side-bet worth half his initial bet on the outcome that the dealer has a blackjack | Available if the player's hand value is less than 21, the first card of the dealer is an Ace, and |

| | | |
|--|--|-------------------------------------------|
| | | the player only has his initial two cards |
|--|--|-------------------------------------------|

Playing double or insurance further requires the player to have a sufficient balance to cover the increased bet size.

After all of the players have finished their turn, the dealer plays in a predetermined manner: He draws cards as long as his hand is worth less than 17 points and must stand on a soft 17 or better.

Evaluation

Once the dealer completes his turn, each player's hand gets *evaluated* against the dealer's hand. As described above, the goal of each player is to get a hand total closer to 21 than the dealer without busting. Concretely, a player wins by achieving either of the following final game states:

- A value higher than dealer's without exceeding 21
- Any value less than 21 while the dealer busts

Winning players get paid even money, i.e. 1:1 on the initial bet. If they win with *blackjack* (hand value of 21 with two cards), they receive one and a half times their bet (3:2). Furthermore, in case a player has decided to place the insurance bet and the dealer has indeed blackjack, this bet is paid out 2:1. A player winning a \$5 insurance bet receives back the \$5 plus an additional \$10 from the bank, for example.

If player and dealer have equal hands and did not bust, they tie, and the player's bet is returned (also called *push*).

In any other case, e.g. the player busting even if the dealer busts as well, the player loses their bet.

Architecture

As our blackjack game is implemented as a web application, the basic architecture resembles a client-server model, where the client (front end) runs in a web browser and directly communicates with the server (back end). Due to the simplicity of our application and the utilization of XStack, we resided from using more sophisticated architectures, such as Client-Server-Broker or Client-Server-Dispatcher.

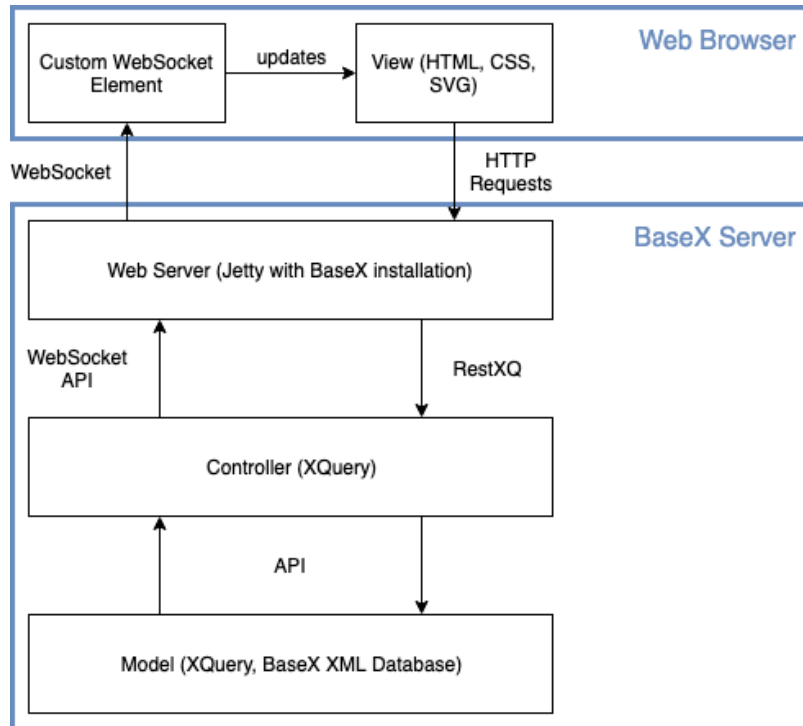
The server-side Business Logic and Persistency Layers are realized using the XQuery processor and XML database of BaseX. The content is rendered on the server-side using an XSLT processor.

Communication using only HTTP requests and responses is not bidirectional. The client can send stateless requests that are answered by the server with a response. The server is not able to send requests to the client, as no active connection is maintained after fulfilling a request. This kind of communication is sufficient in web applications where the business logic is only based on the actions of the client. Applications in which the server should be able to update the content of the client without any requests require full-duplex communication. Examples for such applications are chat rooms and multiplayer video games. Bidirectional communication can be realized using web sockets. Here, a Transmission Control Protocol (TCP) connection between the client and the server is established and maintained until the client or server closes the connection. Therefore, the server can send new content to the client without incoming requests until the connection is closed. BaseX supports Web Sockets, utilizing the Jetty's WebSocket servlet API. As we decided to implement a distributed version of the game, allowing multiple clients to play together, we made use of the WebSockets support of BaseX.

Further, our app complies with the Model View Controller (MVC) architecture to enable separation of concerns and increase the maintainability of the application by decoupling the data representation from the data access. In MVC, the system is separated into 3 different subsystems: the model representing the data access, the view for data presentation and the controller as a mediator between the data presentation and access. Implementations of MVC can be categorized as Push and Pull variants. In the Pull variant, the data is retrieved by the view from the model, while in the Push variant the model updates the view after changes to the data. In our application, every player should see the current state

of the game. Therefore, we chose the push variant. The resulting architectural structure of our web application can be seen below.

Figure 2. Model of the Resulting Architecture.



The component model

Modeling the model

In an MVC-architecture the model has the control of the application's logic. Furthermore, it is responsible to store data, for example in a database. In order to maintain data, several objects with unique functionality are needed. All these collaborating objects provide different attributes and methods. In the following, the data model will be described first. Thereon the design decisions will be discussed. The second part consists of the implementation of the game, followed by a discussion on implementation details and which problems were faced.

The primary database `xforms-games` consists of a list of games. For that, a class `Game` is needed which stores all information about the instance of a blackjack game. A game object needs an attribute to store its' id and another attribute to store the current state of the game. Regarding `Game` functions, their main purpose is constructing, initializing and resetting game objects. Because every game object also stores all the attending players and a dealer, the functions essentially execute operations on all players. In addition to the basic functions, `Game` must provide some function to evaluate the game. However, the logic of evaluating a player should be sourced to the `Player` class. Another object stored in a game object is the chat.

The `Player` will need some attributes to store metadata like name, balance, bet, and profit. Furthermore, this class has to hold a player's state and whether or not a player chooses to take insurance. One more object stored inside the player is the player's hand, which holds the cards. As to the `Player` functions, contiguous to the constructors and setters, there will be a lot of functions that contain the game's logic. Depending on the game's state, a player can take several actions, which will be passed over to the `Player` class by the API. Alongside the playing functions, some helpers which provide logic for finding the next player and joining or leaving a game are needed. Finally, logic to evaluate a player must be integrated.

Since the players all play against the dealer, another class `Dealer` is required. It does not require any attributes, but it must store the deck of the game and a hand object similar to the player's. Concerning the functions of the class, it should provide a helper function to help to determine a player's profit, depending on their insurance state. More logic to give every player two cards and also draw for the dealer will be required.

The next class we need in our model is `Hand`. On the one hand, this class will provide helper functions for evaluating a player's hand. On the other hand, it will supply some function to calculate the optimal sum of a hand. This is needed because the card Ace can have the value 1 and 11.

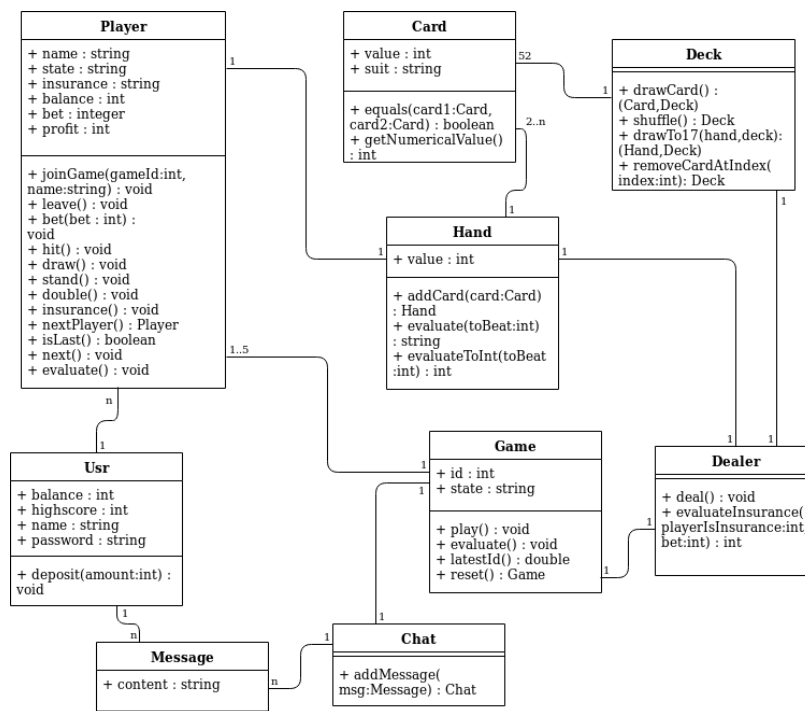
The `Card` class will provide two attributes to store a card's value and suit. Besides the constructor and setter functions, there will be a getter function that returns a card's actual value,

Inside the `Chat` class, the players' messages should be stored.

The last class in the model component is `Usr` (`User` is a reserved namespace in BaseX). It has no functions other than a constructor and setters. Users are stored database "xforms-users".

The model is represented in the following class diagram. Helper functions and setters/getters/constructors are omitted for readability.

Figure 3. Class diagrams for component Model.



Design decisions and discussion

Continuing the section about the modeling of the model, all the design decisions will be presented and discussed.

One of the decisions was to implement a class that takes care of the players' and the dealer's cards because we did not like storing single card instances inside a player or dealer element. We came up with the `Hand` class which is holding cards for the players and the dealer. It has the benefit of getting their cards summed up value easily, as well as evaluating a player's hand. Another great advantage of implementing this class is that the feature "split" could be added to our solution without the need to change other classes. In the real worlds game of blackjack players can decide to split their initial hand which allows them to play with two hands at one time. Without the hand class, some other ideas

would be needed to enable this feature. For example, one could add another attribute to the `Card` class which indicates to which hand it belongs.

Because we want to store the global balance of a player permanently, we introduced the `Usr` class. This way we can easily look up the player's highest achieved balance for the highscore list in the lobby. For every game a `Usr` joins, a new `Player` instance will be created with the balance of the `Usr`.

We choose to have the `Deck` as an extra class so that we can simply store all of the game's deck's cards in one single instance, instead of having all of the cards inside the `Dealer` or `Game` object. This grants a better structure in the model and simplifies functionality like shuffling the deck or drawing cards. When thinking about extending the game, this class can be helpful for example when one wants more than one deck. (in some variants of blackjack, each player has a separate deck)

So why do we need `Dealer` instead of just having another player object? At first, both classes may not differ a lot looking at what players and the dealer do in a game of blackjack. However, by closer inspection, one notices that the dealer has a lot of differences. Starting with the attributes, a dealer does not need any meta information stored like a name or balance and the dealer does not place a bet each round either. The only resemblance is that both objects store a hand object. Comparing the functions of the classes, the dealer has no choices to make, so none of the player's functions that are needed for a player's decision are required. Without the `Dealer`, we would need another way to store the cards included in the game. Altogether a dealer object is too unequal to just use another player object instead.

Another question is how to deal with players that lost all their money? We decided that those players get set to spectating mode and can deposit more money in the main menu

Since we built a multiplayer game which, in principle, could be played by clients in distributed locations over the internet, we needed a way to let players communicate with each other. We therefore decided to implement a chatting feature to make the experience more social. Moreover, when players leave or join the game a small information text will be inserted into the chat so other players will know that someone joined/left their game.

In conclusion, our main idea while modelling the model was to prevent writing the same code multiple times and segregating code by its functionality. Instead of having a few huge classes with a big range of functions we rather wanted to have more but smaller classes that are not too complex. This way the model stays a lot more readable and it makes it more adjustable and integrating extensions easier.

Implementing the component Model

The component model has the responsibility of holding data. Since we use an object-oriented approach, all implemented classes provide their own functionality and attributes. So each class contains data which is altogether stored in our `BaseX` database. The database also lets us query functions via `XQuery` and the `XQuery-Processor` because of that all of the component model's functions are written in `XQuery` and all the data is stored as XML elements. `XQuery` has a functional approach and was created to run queries on XML databases. One of the most important concepts of the language are `FLOWR-expressions`. The name `FLOWR` contains the initials of "for-let-order-where-return" and can be regarded as the equivalent of "select from where" queries in `SQL`. The functions need an additional parameter `self` which indicates on which object the code should be executed to emulate object-oriented programming in `XQuery`.

Because most of the classes have setters, getters, and constructors and the implementation of these functions do not differ in comparison to other classes we don't describe the implementation for every single class. In fact, all setter methods return a new element by calling the constructor with alternated parameters. All of the constructors create XML elements depending on the attributes of a class.

Beginning with `Game` class that has the following attributes and functions.

Table 2. Game attributes

| <i>Name</i> | <i>Type</i> | <i>Description</i> |
|-------------|-------------|--------------------|
|-------------|-------------|--------------------|

| | | |
|-------|--------|------------------------------|
| id | double | to distinguish games |
| state | string | stores the state of the game |

Table 3. Game functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------|------------------|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| evaluate | caller:integer | void | sets game's state to "evaluated", calls player:evaluate for all players, uses caller to determine what action the last player took |
| latestId | none | double ??? | returns the highest id of all games in the database, needed for new games since the id always gets incremented |
| play | none | void | sets game's state to "playing", sets all players besides the first player to inactive, the first one to active and finally calls dealer:deal |
| reset | none | XML element of game | sets game to state "betting", basically calls the classes constructor with all the players that have a balance > 0 |

Table 4. Player attributes

| <i>Name</i> | <i>Type</i> | <i>Description</i> |
|-------------|-------------|-----------------------------------------------|
| balance | integer | store the balance of player |
| bet | integer | stores bet of a player |
| insurance | string | stores whether or not a player took insurance |
| name | string | stores name of player |
| profit | integer | stores profit of a round |
| state | string | stores state of player |

Table 5. Player functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------|------------------|---------------------|-------------------------------------------------------------------------------------------------------------------------|
| bet | bet:integer | void | updates self's bet by given parameter bet, calls player:next |
| double | none | void | updates the player's bet to two times of the amount of bet, calls game:evaluate if player is last otherwise player:next |

| | | | |
|------------|--------------------------------|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| draw | none | void | calls deck:drawCard and hand:addCard to add a card to the player, updates the player's hand and the deck |
| evaluate | none | void | calculates profit of player and updates self's balance accordingly, also updates user's balance accordingly |
| hit | none | void | calls helper player:draw, gives player another card and if they bust calls player:next or game:evaluate if they are the last player |
| insurance | none | void | updates self insurance to "true" |
| isLast | none | boolean | determines and returns whether player is last |
| joinGame | gameId:integer, name:string | void | if it is the first player, creates a new game object with the new player, else just inserts the new player and adds an information to the chat about the player joining |
| leave | none | void | deletes self and calls player:next if self was active and there is more than one player, adds leave information to chat |
| next | none | void | sets self state to "inactive" and if there is a next player:nextPlayer to "active" else checks if game's state is "betting" then calls game:play otherwise game:evaluate |
| nextPlayer | none | XML element player | determines and returns the next player |
| stand | none | void | calls player:next |

The dealer has no attributes.

Table 6. Dealer functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------|------------------|---------------------|-------------------------------------------------------------------------------------|
| deal | none | void | firstly calls deck:drawTo17 for self, afterwards deals two cards for each player by |

| | | | |
|-------------------|---------------------------------------------------------|---------|-----------------------------------------------------------------------------------------------------------|
| | | | adding cards of the deck to hand and removing them from the deck |
| evaluateInsurance | playerIsInsurance:integer,bet:double,player-Won:integer | integer | calculates a player's bonus depending on whether they have an insurance and if the dealer got a blackjack |

The deck class has no attributes.

Table 7. Deck functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------------|---------------------------------------------|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| drawCard | none | (XML element card,XML element deck) | calls deck:getCard to get the topmost card and calls deck:removeCardAtIndex to remove topmost card to return the new deck |
| drawTo17 | hand:XML element hand,deck:XML element deck | (XML element hand,XML element deck) | recursively draws cards from given deck to given hand until hand's value ≥ 17 |
| removeCardAtIndex | index:integer | XML element deck | calls helper deck:removeCard that finds and removes the card at given index and returns a new deck without the card |
| shuffle | none | XML element deck | uses a random number generator to create a new deck with a random permutation of the cards |

Table 8. Hand attributes

| <i>Name</i> | <i>Type</i> | <i>Description</i> |
|-------------|-------------|-------------------------------------------|
| value | integer | stores the sum of the hand's card's value |

Table 9. Hand functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|---------------|-----------------------|---------------------|--------------------------------------------------------------------------------------------------|
| addCard | card:XML element hand | XML element hand | adds the given card to given hand and returns new hand |
| evaluate | toBeat:integer | string | checks if player beats the dealer and returns a string accordingly |
| evaluateToInt | toBeat:integer | integer | checks if player beats the dealer and returns an integer accordingly, needed for player:evaluate |

| | | | |
|---------------|-----------------------------|---------|--------------------------------------------------------------------------------------------------------------|
| getOptimalSum | values:sequence of integers | integer | recursively calculates optimal sum in favor of player, replacing aces with value 1 if the sum of values > 21 |
|---------------|-----------------------------|---------|--------------------------------------------------------------------------------------------------------------|

Table 10. Card attributes

| <i>Name</i> | <i>Type</i> | <i>Description</i> |
|-------------|-------------|------------------------------|
| suit | string | stores the suit of the card |
| value | integer | stores the value of the card |

Table 11. Card functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------------|------------------------------------------------|---------------------|--------------------------------------------------------|
| equals | card1:XML element card2:XML element deck | boolean | checks if value and suit of both cards are equal |
| getNumericalValue | none | integer | returns cards value, 10 for picture cards, 11 for Aces |

Table 12. Usr attributes

| <i>Name</i> | <i>Type</i> | <i>Description</i> |
|-------------|-------------|--------------------------------------------|
| balance | integer | store balance of user |
| highscore | integer | store the all time highest balance of user |
| name | string | stores name of user |
| password | string | stores password of user |

Table 13. Usr functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------|------------------|---------------------|------------------------------------------------------------------------------------------|
| deposit | amount:integer | void | updates user's balance and replaces user's highscore with balance if balance > highscore |

The chat has no attributes.

Table 14. Chat functions

| <i>Name</i> | <i>Parameter</i> | <i>Return Value</i> | <i>Functionality</i> |
|-------------|------------------|---------------------|-------------------------------------------------|
| addMessage | self,msg:string | XML element chat | adds given message to chat and returns new chat |

Implementation details, decisions and discussion

While implementing the model, we faced a few problems and the “XQuery-Update-Facility” because XQuery is “READ-ONLY”. In this section, some examples will be illustrated where we had to rethink

our attempts of implementing database updating logic. Also, some interesting parts of our code will be discussed.

The first problem we faced was that one cannot replace the same node in one return statement more than one time. Because we want a comfortable game that requires as few inputs from the user as possible, we make functions like `player:hit` automatically call `player:next` when the player busted. That way the busted player stands and the next player can play without any further user input needed. So in our first attempts of implementing the dealer draw function, we ran into the mentioned problem because if the last player calls hit (which replaces the node of the deck) and busts at the same time, the `dealer:draw` function is called which also will try to replace the deck. A quick workaround was found by not updating the deck in the dealer draw function since after the dealer has played the round is over and we don't need the deck anymore. However, we decide to have the dealer draw at the beginning of a round to avoid running into this problem in the future.

As to the second problem, which has the same root as the first problem. Since `player:double` automatically calls `game:evaluate` if the player was the last one, the database was not updated yet so `evaluate` would still operate on data pre the `player:double` call. This means that the last player's bet is not doubled and their hand is still their initial hand. One solution to this could be to have another game state like "toEvaluate" after all players have played and the game is ready to be evaluated. However, at least one user input to have the game evaluated would be needed which interrupts the game flow. This problem also occurs when the last player busts. Our final solution is to add a parameter `caller` to `game:evaluate` so that the function knows which action was just executed. In case that `caller` equals 0 the last player decides to stand so no updates to the database are required. When `caller` equals 1 the last player busts, so we need to make his profit negative. The third case is when `caller` equals 2, this means the last player doubles so we need to add one card to their hand and doubling their bet before evaluating the player's hand.

For the `Deck` implementation, we decide to create a "realistic" deck since we don't like the idea of generating random cards each time a draw occurs. Randomly generating cards would either face the problems of possibly having duplicate cards or needing to implement some code to check if a generated card already exists and repeat generating a card until it is no longer a duplicate.

Like mentioned before, we want a comfortable experience for the player by always ending one's turn automatically if they choose to double or bust. To achieve this we implement the function `player:next`, `player:nextPlayer` and `player:isLast`. Together these functions check if a player is the last one and help to set the next player's state to "active" otherwise. This can be used in both game states where inputs of all players is required (betting and playing).

Lastly, at the end of each round, when players are informed about their profit, any player can initiate the start of the next round without waiting for the other players. An alternative behavior that we considered was to implement a "Ready-Up"-functionality and, instead, wait for all currently participating clients to confirm that they want to proceed before starting the next round. We finally decided against a ready-up semantic for the following reasons: The final evaluation screen informs the player about his profit with only a single number, so there is no reason for the user to stay on this screen for a longer period of time. Nevertheless, we found that, with a ready-up semantic and multiple participating clients, we noticed that the flow of the game was constantly disrupted and players spent more time waiting for others to ready up than playing. Also, users that do not want to take part in another round can leave the table at any time during the betting or even the playing phase. We, therefore, found that initiating another round by any player was sufficient and creates a smoother experience overall.

The component View

Modeling the component View

The view within the MVC architecture is used to display different kinds of screens. The basic structure is divided into Lobby and Game. While the Lobby is used for basic interaction of administrative nature, the Game displays the actual playing environment. The current view, as well as the interaction of the view, is specified by the current path which is defined within the API controller (see Controller section). The following table lists all screen types with the corresponding information and interaction.

Table 15. Model for View

| <i>Screen</i> | <i>Information</i> | <i>Interaction</i> |
|--------------------------------------------|---------------------------------------------------------------|----------------------------------------------------------------|
| xforms-multiclient | | login go to signup |
| xforms-multiclient/signup | | create go to signin |
| xforms-multiclient (logged in) | name balance | new game join game highscore logout |
| xforms-multiclient/games | games(id, state, players) | join game new game go back logout |
| xforms-multiclient/highscores | highscore(name, balance) | go back logout |
| xforms-multiclient/games/[id] (join) | depending on game state | join leave chat |
| xforms-multiclient/games/[id] (betting) | balance chat | bet leave chat |
| xforms-multiclient/games/[id] (playing) | cards bets players balances hand values chat | game action(stand, hit, double, insurance) leave chat |
| xforms-multiclient/games/[id] (playing) | cards bets players balances hand values result | new round leave chat |

Implementing the component View

The corresponding XSL-files for Lobby and Game create an XHTML page to display the application in a Browser. As already stated before, the created page depends on the current path but also the current state of Game and Player.

Based on the restriction to XHTML, we frequently use HTML tags such as button, form, input, label, and table. In order to customize the appearance, a global Cascading Style Sheet (CSS) applies to all the created XHTML pages. The style.css file can be found in the static/css folder. We make use of different CSS features such as counters, advanced collectors, transitions and media queries. Through the media queries, we enable a responsive user experience. Depending on the screen size elements can either be hidden like the chat for instance or displayed with a different shape such as the tables. For the general layout, we apply a basic and minimalistic design that enables a uniform appearance across the whole application.

To enable the basic interaction, we use the submit type of the HTML buttons. On click, they create a GET/POST HTTP Request which in turn are handled and specified by the API. Within the Lobby, those buttons represent the central element.

On the starting screen, a player can either log in or register for the game. Once logged in, his current balance is displayed as well as the different buttons to create a new game, join an existing game, to see the highscore or to reset his balance. While the new game button will simply create a new game, the join game and highscore buttons lead to another screen including an HTML table with corresponding information. The Player can join a displayed game by clicking on the corresponding row.

The core element of the game is the table, which is implemented with SVG. Depending on the current situation, the SVG includes corresponding cards, chips, and labels(including player name, balance, and hand value). Our graphics are hierarchically structured to avoid redundancy. The graphic for a specific card (e.g. the seven of hearts) is a specific instance of a more general *suit card* (e.g. the hearts card), which in turn is based on the empty *base card*.

Concerning the game interaction, we implemented a dialog box placed below the table. A player can join the game, place bets or make context dependent actions like hit, stand, double or insurance. The dialog box is also used to display the results of a round, informing the player about his wins or losses. In the bottom left-hand corner, the chat is displayed. In order to display/hide the chat window, an intuitive toggle mechanism was implemented. In the top left-hand corner, a player always has the ability to leave the game and to go back to the lobby.

The component Controller

Modeling the component Controller

The controller has the task to mediate between the data representation (view) and the data access (model). Therefore, the controller has to handle all requests of the view concerning the game, but also the establishment and teardown of the connection.

Implementing the component Controller

Possible requests from the client are: menu, signup, login, logout, setup, games, highscore, delete, join, leave, draw, bet, hit, stand, double, insurance, new round and chat. These are either forwarded directly to the model or answered with static Html files. The requests menu, signup, login, logout, highscore are used for the account control and the displaying of the highscores. The remaining requests are used to implement the game functionalities, including the group chat.

In XQuery functions are limited to either updating or querying the database. This is a problem when trying to modify objects and return them afterwards. The workaround, we use for our implementation,

is the BaseX specific function `update:output()`. This function returns the results of a query after all updates have been applied.

Collaboration

The implementation of the web application was done as a group project with a group size of 4. Git was used as a distributed version control system. Further, the Github Issues feature was used to track and assign tasks and problems. Different branches were utilized to test different possible implementations.

Regular meetings in person at the Informatik Faculty of the TUM and Voice over IP chats (TS3) were arranged using the messenger whats app regularly to coordinate the collaboration and progress. Visualizations of git concerning the issues and commits can be seen below.

Figure 4. Git Overview (Master).

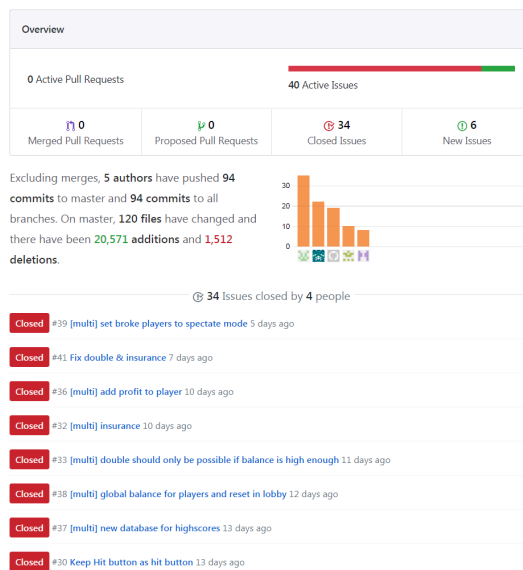


Figure 5. Code Frequency (Master).

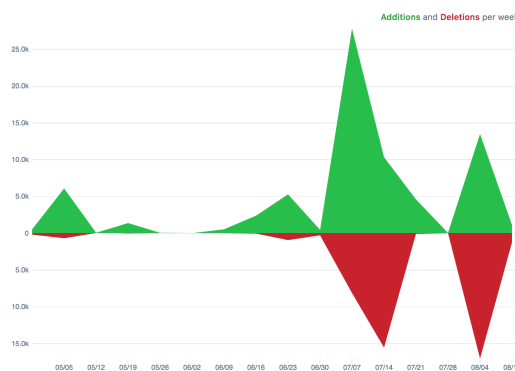
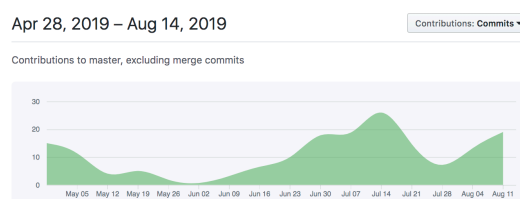


Figure 6. Contributions (Master).



Conclusions

We have described the implementation of a multi-client capable web application using XStack technologies. Starting from a domain model of the game blackjack, we used a domain-driven design approach to convert our conceptual model into a functional program.

Possible next steps for this project would be the deployment of the system to the public internet and the integration of different or more extensive rule sets, e.g. the implementation of the described split move available in some variants of blackjack.

Materials and support required for the implementation of the black jack game are all provided by the practical course over moodle. Additionally the exercises are a good guidance on how to implement the game. Further personal support is offered by the course.

Bibliography

[BaseX] BaseX homepage. <http://basex.org>. Last accessed on 2019 June 16 .

[BaseXStomp] BaseX with STOMP support, main repository. <https://github.com/BaseXdb/basex/tree/stomp>. Last accessed on 2019 August 8 .