X Stack Multiclient Blackjack

Janik Schnellenback <janik.schnellbach@tum.de>

Maximilian Karpfinger <maximilian.karpfinger@tum.de>

Felix Hennerkes <ga38hom@mytum.de>

Daniel Meint <d.meint@tum.de>

Table of Contents

Introduction

Current XML technologies provide a full stack of modeling languages, implementation languages, and tools for web applications that is stable, platform independent, and based on open standards. A particularly strong point of what we call the X stack is that data are encoded with XML end-to-end and that XML technologies can be used whereever XML data need to be processed. Combining principles and proven practices from document and software engineering, this paper documents architecture, modeling techniques, and implementation strategies for the simple browser game Guess the Number (GN).

Description of the case study

We implement the popular casino game Blackjack as a multi-client web application. Globally, there are a variety of rulesets with slight differences. We seek to employ 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*). Players win by achieving one of the following final game states:

- A hand value of 21 with only two cards (*blackjack*)
- A value higher than dealer's without exceeding 21
- A value less than 21 while the dealer busts

Before the actual playing begins, players place their *bets*. Our version does not limit players in the amount they want to bet, but prohitibts them from playing without betting at all.

At the beginning of a round, each player is dealt two cards face up. The dealer receives one exposed card that everyone can see and one *hidden* card. Now, the dealer asks each player, going clockwise around the table, whether they want to improve their hand by drawing additional cards (*hit*).

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.

After all players have finished their turn, the dealer players 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.

The game GN, our case study, has two types of actors: Player and Game. Upon start of a game, Game thinks of a secret number (`secret`) between 1 and some upper bound (`range`). Player guesses repeatedly what the secret number is and receives feedback from Game whether the guess is high, low or correct. There is a limit to the number of guesses allowed (`maxGuesses`) that depends on `range`. Player wins if they guess the secret number correctly within the maximal number of guesses; Game wins otherwise. There are no ties.

We develop a web application that Player calls from a web browser to play GN. The app creates a new Game actor for each initial request and offers a user interface to Player through the browser. The user interface displays information about the state of the game and offers pertinent interactions to Player. The app handles displays for any number of games simultaneously.

In developing GN, we follow a domain-driven design process [E03], starting with a domain model of GN and relating all further design and implementation artefacts to the domain model in a systematic way, so that in the end the implementation still serves as a model for the system. We believe that this process is particularly beneficial in end-user programming, which is common in XML experts developing their own systems in the X stack.

Architecture

Any web application uses by definition a client-server architecture with the following characteristics:

- The client runs in a web browser.
- The server runs in a web server (Apache, Jetty)
- Client and server communicate through HTTP requests and responses.

The implementation of the server typically requires application code, that is run in some servlet or other container on the webserver, and a backend component such as a database system to persist data. Architecturally, the backend component is layered below the web server, and only the web server communicates with the backend component, using some kind of API. This leads to the typical three-tier architecture of web applications with any number of web browsers that are dynamically instantiated on user request in Tier 1, a web server in Tier 2 and a backend system in Tier 3. The layering of the three tiers is guaranteed through the pattern of communication between the tiers. Only the two pairs of Tier 1 and 2 and Tier 2 and 3 communicate, the former through HTTP and the latter through some kind of API. [A14]

Software engineering has settled on the Model View Controller (MVC) architectural style for systems with a user interface [BD13][F12]. For this case study, we decide on an MVC variant called Passive View [F06]. In Passive View, the responsibilities of the component View are elementary. View displays the state of the application as communicated by Controller and offers user interactions on Controller instruction. View notifies Controller of any user interactions and waits for new information from Controller. Thus, View delegates any real processing to the other two components that we run on the server.

View and Controller decide on a data format for their communication. In the X stack, the exchanged data are represented in XML.

The classic benefit of this approach is testability. We choose the Passive View variant so that we can focus our attention on one set of technologies, the ones we use on the server, first.

The component Model holds data and provides access to the data in the form of methods that manipulate the data. Model exposes its functionality through an interface or API. In the most simple cases of web applications, we have a passive component Model and a single component Controller for Model: Methods are called by the Controller of Model; state changes in Model always originate from methods that are called by its Controller.

Controller receives notification from *any* View about user interaction. It interacts with Model to handle the notification, converting information from Model into information for View (which data to display, which interactions to offer) and returning this information to View.

The hallmark of the MVC architectural style is that it separates the user interface, represented by View, from the data and functionality of the application, represented by Model. A number of View components can work with the same Model component, mediated through Controller components, without Model having to know anything about the Views that are connected to it; each View may retain its own methods of presenting data and interacting with users without having to coordinate with any of the other components.

How can we map the three components Model, View and Controller onto the three tiers of a web application? It is tempting to paint with a broad brush and map Views to Tier 1, Controllers to Tier 2 and Models to Tier 3. However, in an XML-based web application like GN, we implement both Controller and Model as XQuery modules that are run by a single backend XQuery processor in Tier 3. Following the MVC architectural style, Controller and Model are layers within Tier 3 that communicate through an API. The web server in Tier 2 acts as a generic component that handles network communication; that is, HTTP requests and responses from web clients. The HTTP requests are mapped to XQuery functions in Controller as defined by RestXQ annotations within Controller. Controller also constructs complete HTTP response data, partially using RestXQ elements, that are then just passed back as HTTP responses to the correct web client by the web server. Thus, Tier 2 runs no application-specific software. Both Controller and Model reside in Tier 3, which has to be split into two layers for the two components.

In our particular implementation we use the XML database system BaseX as XQuery processor. Any other RestXQ aware XML database such as eXist or MarkLogic would work as well. we have chosen BaseX over eXist for its support of the W3C standard XQuery Update Facility, and we prefer BaseX over MarkLogic for no other reason than that it is free.

Figure 1. Passive View mapped to three-tier web application architecture.

In the following three sections, we present a domain model for GN by modeling three components Model, View, and Controller as a series of tables and annotated UML class and state diagrams. We also discuss how to implement each component in the X stack.

The component Model

Modeling the component Model

In this section, we first address the component Model, whose responsibility it is to hold data and to provide access. In principle, we structure the data of the component into collaborating objects that provide access through methods that operate on the specific data of each object on which they are called, applying the *encapsulation principle* of object-oriented system design. The general advantages

of encapsulation are that it makes models and code easier to understand, to maintain, and to extend. In the case of GN, Model manages a repository of game objects which we model as instances of a single class Game.

In the case of the GN component Model, we have come up with the following object attributes of class Game: `id`, `range`, `secret`, `maxGuesses` and `guessesSoFar`, with some type information and constraints.

As to the Game methods, they follow the general scheme of reacting to some kind of event by updating their data, possibly in collaboration with other objects, and returning status.

This requires us to model another class, GameStatus, that presents the status of a Game object as exposed to the environment. GameStatus objects are pure value objects without explicit methods (they do have implicit getter methods to read out the data they carry). GameStatus objects are only passed as values between objects but are not persisted.

We have identified two Game methods. The first, `newGame`, accepts the range of a new game as its parameter. It creates a new game and inserts it into the repository of games, then returns a GameStatus object. The second, `evaluateGuess`, operates on a specific game object and accepts a guess as its parameter. It advances the number of guesses for that game and returns a GameStatus object.

Both methods follow the general scheme introduced above. The first method is a class method in that it does not operate on an existing object. It functions as a constructor, returning a reference to its newly created object as part of GameStatus. The second method is a classical object method, operating on an object and returning a value.

A game object, once created, only offers one method to call, `evaluateGuess`, and there are some constraints to this use of the object: the maximal number of calls to the method is constrained by `maxGuesses` and the method may not be called any more after the secret was once guessed correctly. Hence, the object can be in one of two states, active or over, and its behaviour depends on the state it is in. We model this in a UML state diagram and also represent the state with a new boolean attribute `isActive` in the Game class. This attribute needs to be updated by method `evaluateGuess` in accordance with the state diagram.

The domain model for component Model is represented in the following UML diagrams.

Figure 2. Class diagrams for component Model.

Figure 3. State diagram for component Model.

Implementing the component Model

The architecture of GN shows that the component Model holds the data of the games and that it provides an API to the component Controller. We implement Model using BaseX in the following way:

Game data as defined by the Game class diagram are represented as XML elements named game. Each game element that represents a game object has a unique ID and a subelement for each of the attributes of the object's attributes. All game elements are collected in a container element named games which in turn is a ressource in the database guessTheNumber in BaseX.

Game methods as defined by the Game class diagram are implemented as XQuery methods that use the XQuery Update Facility extension. The methods are collected into their own XQuery module and that are executed by BaseX [BaseX]. Object methods such as evaluateGuess in our example get an ID of the object on which they operate as a first parameter. This parameter plays the role of `this` or `self` in object-oriented programming languages such as Java or Python.

A typical method needs to create or to update object data, which means that it needs to perform database updates, and it needs to return values. Consequently, a typical implementation of a method needs to make use of both XQuery for reading data and for computing new values and the XQuery extension XQuery Update Facility for updating the database.

This confronts us with the update constraint of extended XQuery methods: they can either return a value or a list of pending updates that will eventually be committed to the database. For our implementation strategy, we need a way to do both, return a value and update the database, within a single query, and a way to control the timing of updates, so that methods higher up or later in the call hierarchy operate on up-to-date data.

BaseX provides an extension of XQuery in the form of a method called update:output, with the update namespace prefix defined by default and bound to some specific namespace url. This method allows XQuery methods to return a value that is provided as argument to update:output in the presence of a pending updates list.

There are a number of caveats to this feature: First, the return value depends on the "old" status of the database, before the pending updates are performed. Second, a method that returns a pending updates list can only be called in the return value of a query expression, not, for example, in a let statement that defines an intermediate variable. Finally, and worst of all, the pending updates that a method returns will be merged with pending updates that a function higher up in the call hierarchy produces, seriously messing up the functionality that a method provides.

We can deal with all these constraints if we call a query method indirectly, via HTTP. For the component Model, we annotate the two methods createGame and evaluateGuess with RestXQ URL patterns. Methods that call the Model API, in this case Controller methods, compose a RESTful URI with the URL and parameters of the Model method, send that with the http:send-request method and get an HTTP response element back that has the return value of the Model method in its body. The Model method will have committed its pending updates as a side-effect before sending its response. Note that the method http:send-request is part of the HTTP Client Module that is defined by EXPath, an industry standard. The namespace prefix http is predefined and bound to the URL http://expath.org/ns/http-client.

If a method requires further functional decomposition and the inner methods also need to perform updates, the same approach can be applied: Put a RestXQ annotation with the definition of the method and call it indirectly via HTTP.

There is one final caveat regarding deadlocks that is familiar from concurrent programming and that requires a standard practice, as described below.

BaseX treats each XQuery method as a transaction. It uses the Two-Phase Commit Protocol [BaseX] to guarantee ACID criteria. The granularity of locks is coarse: complete databases or the complete BaseX system are locked.

When a query is executed, BaseX analyses which read locks and which read-write locks the query requires. If it cannot determine the exact databases to be locked, it locks the complete system. A read lock is granted for a ressource if only read locks are currently in use. A read-write lock is granted for a ressource if no other locks (neither read nor read-write locks) are in use. All locks that the query requires must be granted before the query starts executing.

Consequently, when a query issues an HTTP request that updates a database and later wants to read the same database directly, there will be a deadlock. The reason is that the read lock on the database will be granted to the query when it *begins* to execute, and then the inner HTTP request cannot be served since it cannot aquire the necessary read-write lock.

There is an obvious deadlock-prevention strategy: If a query accesses a database via HTTP, it must do all its accesses of that database via HTTP, even read accesses that are executed after write accesses have been executed. That is similar to the practice in concurrent programming: If a system needs to protect a ressource against concurrent access, locks have to be used for both read and write access. It is not sufficient to protect against concurrent write [G06].

In principle, this procedere weakens the ACID guarantees for queries as transactions since database accesses are encapsulated into http:send-request calls and no longer visible on the outside. In the case of GN, the user interface guarantees that calls to the API of Model do not interfere with one another. Calls to class methods and calls to methods of different objects from any View may interleave freely. Since a single game is only played from a specific View, no other interleavings occur.

The component View

Modeling the component View

The component View of GN has a number of screens: welcomeScreen, firstGuessScreen, furtherGuessScreen, resultScreen and goodByeScreen. The domain model for View defines, for each type of screen, which information to display and which types of interaction to offer. It also defines the information that View sends to Controller encoded into HTTP requests. The GN app is started by requesting a game start URI from a web browser.

We define the domain model for View in a tabular format.

Table 1. Domain model for component View

Screen type	*Information*	*Interaction*	*Request*
			guessTheNumber
welcomeScreen		fill in range submit	newGame[range]
firstGuessScreen	id guessesSoFar (static, 0) maxGuesses range	fill in next guess submit	guess[id,guess]
furtherGuessScreen	id guessesSoFar maxGuesses range evaluation last guess	fill in next guess submit	guess[id,guess]
resultScreen	id guessesSoFar maxGuesses range evaluation game secret	play again quit	playAgain quit
goodByeScreen			

Implementing the component View

The component View is implemented as a single XHTML page with embedded XForms components.

The XForms model holds in its main instance the current screen type and the information for the current screen. In two separate instances, it holds the information that needs to be edited in the screen and transfered to the server. There is one separate instance to fill in the range and another one to fill in the next guess. The latter copies the id of the current game from the main instance since that needs to be retransmitted back to the Controller component, which is stateless and handles any number of

games concurrently. The copying accomodates the fact that an XForms submission can only submit data from a single instance.

The XForms model also defines all submit actions that GN requires. A submit action triggers a GET or a POST HTTP request for static or dynamic requests, respectively. A POST request submits the appropriate instance in the body of the request. Each response replaces the main instance with the HTTP response data.

In effect, Controller sends XML elements to View that describe the data that specify the type of screen and the information that View is supposed to display next. This information is structured according to the following class diagram; the specific information that is to be displayed for each screen type is specified in the domain model for component View.

Figure 4. Class diagram for View data

The body of the XHTML page holds a section for each screen type with XForms widgets that interact with the XForms model. Information about the current state is displayed in a table using XForms output widgets; user input is accepted through XForms input widgets and buttons that trigger XForms submissions. Only the screen type area that matches the main instance's current screen type is visible. The XForms model has a helper instance with a CSS attribute "display: none" that is dynamically read into each section that is inactive.

A more graphical variant of View uses XForms widgets linked to the same XForms model and includes them into an SVG graphic. The widgets are included into the SVG code as HTML-encoded foreign objects that can be styled through CSS and positioned and transformed through SVG. In this variant of View, there are no direct representations of the conceptual screens. Instead, the widgets themselves know when to present themselves depending on the information in the XForms model.

Below, we include a screenshot of the two variants of component View side by side.

Figure 5. Two variants of component View

The component Controller

Modeling the component Controller

Controller is a stateless component that mediates between arbitrary View components and the component Model that it is connected to. The component is required to handle the parameterized requests from a View that are listed in the View model, to call appropriate methods of the component Model and to respond to the View's request with information about the screen type and data that the View is to use next.

Implementing the component Controller

The component Controller needs to handle all requests that a View might send. According to the model of View, these are guessTheNumber, newGame, guess, playAgain and quit. Controller has XQuery methods for each of these requests that are annotated with RestXQ annotations. Only newGame and guess require interaction with Model. The other requests can be handled by returning a static file.

For the requests newGame and guess, Controller composes a URI from the request data and calls the API of Model via HTTP. It then analyzes the new game status that the API call returns and transforms it into the format that the View requires. The transformations are simple. They involve renaming of the container element and inserting the correct screen type for the View to use next in accordance with the data for View modelled above.

Conclusions

In this paper, we have described an implementation of a simple case study in the X stack on a technical level, starting with a domain model that we have then continuously refined into code.

We have worked with a Passive View architecture, encoding View data as XML documents. We have demonstrated how XForms and SVG can be used to implement View components that interact with these data.

A larger revision would be to develop a more intelligent GUI that keeps track of the possibilities that are still open for the secret number and perhaps even suggests the next guess to the user. The necessary computations in the client can probably be programmed within XForms. Alternatively, we wish to explore SaxonJS for this feature of GN.

Bibliography

[A14] Adamkó, Attila. "Internet Tools and Services". Lecture notes. https://gyires.inf.unideb.hu/GyBITT/08/index.html. Last accessed on 2019 February 20.

[BaseX] BaseX homepage. http://basex.org. Last accessed on 2019 February 25 .

[B16] Brüggemann-Klein, Anne. "The XML Expert's Path to Web Applications: Lessons learned from document and from software engineering." Presented at XML In, Web Out: International Symposium on sub rosa XML, Washington, DC, August 1, 2016. In Proceedings of XML In, Web Out: International Symposium on sub rosa XML. Balisage Series on Markup Technologies, vol. 18 (2016). https://doi.org/10.4242/BalisageVol18.Bruggemann-Klein01.

[BD13] Brügge, Bernd and Allen H. Dutoit. "Object-Oriented Software Engineering Using UML, Patterns, and Java." Pearson 2013 (Kindle Edition).

[BMRSS00] Buschmann, Frank, Regine Meunier, Hans Rohner, Peter Sommerlad, and Michael Stal. "Pattern-Oriented Software Architecture, A System of Patterns." Wiley 2000 (Kindle Edition).

[E03] Eric Evans. "Domain-Driven Design: Tackling Complexity in the Heart of Software." Addison-Wesley 2003 (Kindle Edition).

[F06] Fowler, Martin. "Development of Further Patterns of Enterprise Application Architecture." https://www.martinfowler.com/eaaDev/. Last accessed on 2019 February 20.

[F12] Fowler, Martin. "Patterns of Enterprise Application Architecture." Addison-Wesley 2012 (Kindle Edition).

[G06] Goetz, Brian. "Java Concurrency in Practice." Addison-Wesley 2006 (Kindle Edition).