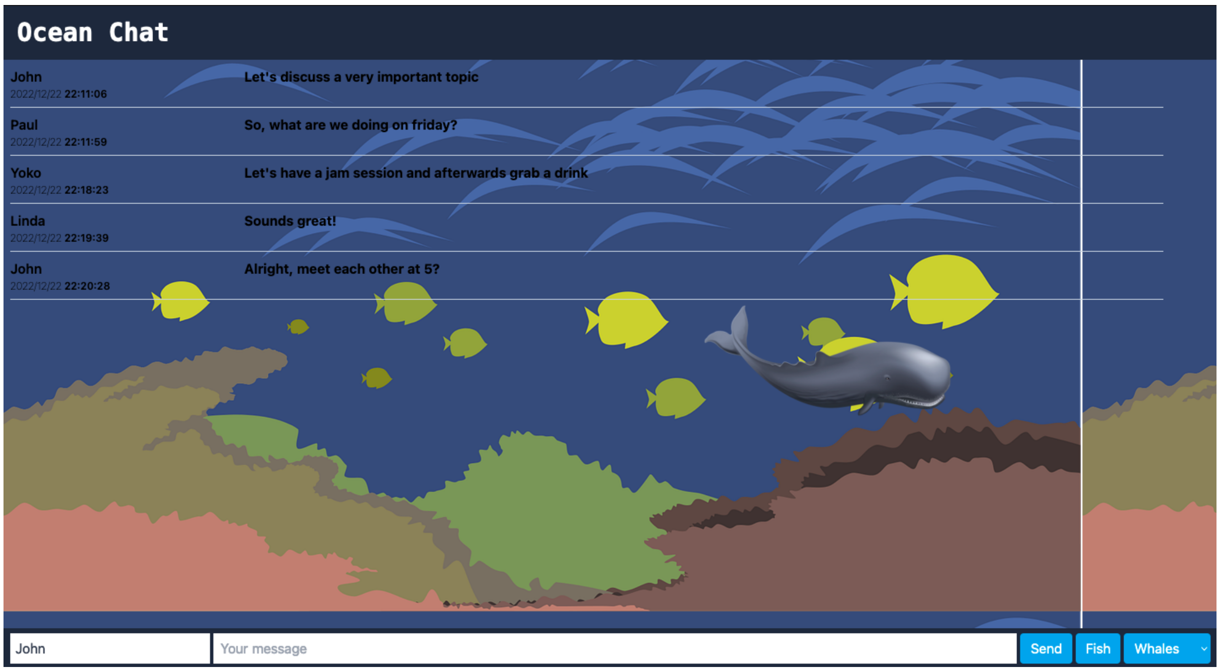
Multiuser Chatroom App



John 2022/12/22 **22:11:06**	Let's discuss a very important topic
Paul 2022/12/22 **22:11:59**	So, what are we doing on friday?
Yoko 2022/12/22 **22:18:23**	Let's have a jam session and afterwards grab a drink
Linda 2022/12/22 **22:19:39**	Sounds great!
John 2022/12/22 **22:20:28**	Alright, meet each other at 5?

Project Group 11

Group members: *Geraldine Hürzeler, Damian Kobler, Tishana Suthenthiran*

SA-2022

https://github.com/damiankobler/Ocean.git

Course: IN.5022 — Concurrent and Distributed Computing

Table of contents

Initial description

We plan to create a chatroom application where multiple users can chat with each other in one chatroom. To do this we want to use Elixir and probably the Phoenix Framework. We implement basic functionality like texting and sending images. The essential problem of a messaging application will be the group communication problem. When having implemented basic functionality, we could imagine implementing further features like having multiple chatrooms, implementing a search function or other functionalities.

Problem statement

These days, many people use chat applications to communicate with team members, friends, and family via their smartphones. This makes these messaging applications an essential medium of communication. Chat apps make an annual appearance in lists of the most downloaded apps. These apps are now deeply integrated in our daily work and personal lives.

Chat rooms have become a popular way for a set of people with common interest to discuss about a common topic of interest. Chat rooms extend one-way messaging between two people to multi-way messaging among a bunch of people. In this project we will implement a simple text-based chat client/server application using state-of-the-art tools in Elixir.

WhatsApp is built with Erlang and Discord is built with Elixir and some Rust. These apps have well over a million concurrent users and need a language that supports concurrent systems. Elixir is therefore a good way to build chat apps.

We want to create a real-time one-on-one browser chat application with a database functionality. The messages should be stored in a database, so that when closing the browser session and reopening it, the messages would not be lost. Ideally, we want the user to be able to connect to different chatrooms and discuss different topics of interest. If possible, we also plan to send images to the user and not only text messages. We want to customize our chat application by introducing some colorful features like sending fishes around.

State-of-the-art

Phoenix is a good platform to get started with regarding chat apps, because it is written in the Elixir language. Elixir is built on the Erlang Virtual Machine, therefore it is made for highly available and highly concurrent applications by nature.

Phoenix is a web development framework, which implements the server-side MVC pattern. Phoenix provides high productivity for developers and high performance for the application, because it uses the advantages of the Elixir language to enable highly interactive server-rendered content. This means interactive webapps can be built without writing a single line of JavaScript.

Phoenix has some interesting features like channels for implementing real time functionality for applications and precompiled templates.

Channels enable real time communication between a lot of connected concurrent clients. It is a persisted connection between the browser and server. The concept of a channel is, the client connects to the server using some transport like a Websocket. When they are connected, they join one or more topics. Each topic has a chat room.

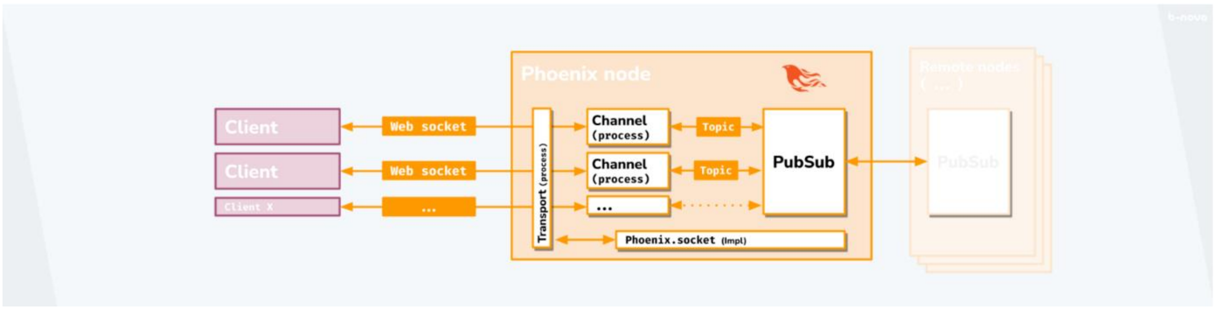


Figure 1: real time broadcasting with PubSub

Clients can send messages to the topics, so to the chatrooms, and can also receive messages from the chatrooms. The same thing happens on the server side. The server receives messages from their connected clients and sends messages to the clients. It’s like the browser and server were on a phone call and there is a constant exchange of information, while both are listening. Because in Phoenix we have Channels based on websockets, the server can send messages as long as the browser maintains the connections.

Servers are able to broadcast messages to all clients in a chatroom. This is Phoenix’s built-in PubSub (publish/subscribe) functionality. (We will talk about this more in the implementation part, since we have used channels with topics and subtopics to open multiple chat rooms.

Precompiled templates: Most of the code in the template files is HTML with embedded Elixir code (files with extension .heex). Phoenix compiles and evaluated these files, which makes everything go a lot faster.

Advantages of Elixir and Phoenix:

1. Elixir and Phoenix apps need little resources regarding memory and CPU, this is extremely beneficial when you are programming in the cloud.
2. Elixir/Phoenix apps restart very quickly, which minimizes downtime.
3. The Elixir / Phoenix ecosystem is quickly evolving and maturing and thus will bring many more features

Presentation of the solution

We were searching a long time for useful tutorials on how to do a chat application in Phoenix. Unfortunately, most of them were depreciated since Phoenix evolves very fast. Finally, we found one which was updated not so long ago and inspired ourselves by this tutorial.

After installing the Phoenix framework and the PostgreSQL Database Server to save the chat messages, the project is ready to be executed.

For creating the application, the directory structure and project files we used the command: mix phx.new chat and fetched and installed all dependencies.

Next, we made use of Phoenix's channel feature and created the first channel called Room using Websockets: mix phx.gen.channel Room

This created two new files in the channels directory, and as explained above in the state-of-the-art part, the room_channel.ex file is for receiving and sending messages.

We then took care of the html part and modified the template files under template/page so that it gives us a nice layout. We implemented two text fields, two buttons for sending the message and sending the fishes, the head of the page and the space for the messages to appear. For the formatting we used Tailwind CSS, since it is an easy approach.

We decided to implement some additional functionality, like sending fishes around. The button "Fish" would make a fish appear on your screen and swim by. Unfortunately, we were not able to broadcast the fish onto other chat members, because we would have needed to create html on the fly, which was too time-consuming and complicated for our level of programming.

To add the WebSocket functionality, we had to uncomment import of socket.js in assets/js/app.js. In this file we made sure that the socket connects to the chat "room" and implemented the eventlistener, that sends the message to the server on the "shout" channel. When the message arrives to the shout channel, it will create a new list element with the corresponding message and append it to the list (this would then be the chat itself).

We decided that we wanted to save the chat history, so by reloading the browser page, the chat would still be there. We created a new data base schema with mix phx.gen.schema and the database table messages, where the name of the user and the message is stored. In our PostgreSQL GUI pgadmin, we were able to see the table schema. In the room_channel.ex file we had to add the changeset in the handle_in/3 function and make it insert into the database with

```
Chat.Message.changeset(%Chat.Message{}, payload) |> Chat.Repo.insert
```

Finally, we had to add some code so that when a new user joins, he would see the existing messages. For that we added a new function handle_info/2.

Now we were able to open two browsers. When sent a message from one browser it also appeared on the other browser page.

After we created the basic chat functionalities, we discussed ways to implement multiple chatrooms that each have their own messages. In the end we decided to do it with Phoenix channel subtopics that use the same relation in the database. So, we created three new subtopic join functions in room_channels.ex using room:chat1, chat2 and chat3. The subtopics handle all the messaging that happens between the clients that are in the respective chatroom/subtopic. This is done with the handle_in/3 and handle_info/2 functions. Upon joining a chatroom, a database query is made to get all the messages related to this chatroom. In the database we added a new attribute called chatroom which has integer values that represent the chatroomID. When creating a new message, the chatroomID corresponding to a chatroom will get saved with the name, message and timestamps.

On the client side we added an index/welcome page and migrated the chat to /chat. The user can choose between chatrooms with a drop-down menu. In app.js we implemented how the client changes from one chatroom to another by joining the room channel and subtopic through the socket. Here we also added the functionality of storing a message with the corresponding chatroomID.

To provide a better user experience, we added the feature of showing a different marine animal based on the name of the chatroom. For example, in the Sharks chatroom there is a shark that swims by instead of the fish we had before. This is mostly done through the JavaScript function sendFish in root.html.heex which will look at the chatroom the client is currently joined to and adjust the image.

Next, we did some research on how to send images in a chatroom developed with phoenix. The only helpful resources here were the documentation of pheonix and ecto. With that it was possbile to create an upload button in the index.html.heex file using a multipart form and the “file input” tag. This step worked perfectly fine. Now we have to process the input the same way as the message input. A blog post showed us that it is possible to save images in the database when working with phoenix and PostgreSQL. So, we decided to go for a solution where we save the images to a new table in our existing database. For the functionalities we can copy the classes needed for handling the messages and modify them for images.

When executing this approach, we got an error concerning the changesets. When modifying our method, we still could not manage to get the use of changesets right. And the time we had was not enough to fully understand how “ecto” and therefore changesets exactly work. That is why no code in relation with the sending image is included in the final solution.

Validation/evaluation of the solution

Testing

At several steps while creating the application we did some testing.

We used the command *mix test* in order to test the html response of the controller. As we changed the index.html.heex file, the assertion with the html response failed. We had to change the assertion to our app name.

Phoenix itself has created some tests: “ping reply”, “shout broadcasting to room”, “broadcasts being pushed to client” and “:after_join sending all existing messages”. We found this very surprising, and this motivated us to use phoenix in further projects.

Before handing in the project, we tested the database, if we added a user and a message if in pgAdmin it would add the message to the database table. Additionally, we deleted messages from the database table and tested, if these messages vanished from the chat as well.

Evaluation

Some important aspects of concurrency and distribution and the problems that go with that are actually handled by the Phoenix Framework which makes development a lot easier for us. The most important server-side Elixir implementation is in room_channel.ex. There, most of the handling and sending of messages between the processes and clients is done. Also, the database query and channel:subtopic joining is defined in this file. Apart from room_channel.ex there are multiple Elixir files that handle a specific part implemented by the Phoenix Framework. We have a router.ex and page_controller.ex, which handle the routing, loading and rendering of the html sites. In user_socket.ex the different channels and their handling through the socket is done. Additionally, there are multiple files that belong to the management of the database access and configuration.

Discussion of the results

Our minimal requirements have been fulfilled. Messages can be sent from one user to multiple other users. Also, a user can choose one out of different chatrooms, in which one specific topic can be discussed. Furthermore, we added a bit color to brighten up the day of each user.

The biggest problem we faced was a steep learning curve of the Phoenix Framework since as Business Informatics students we never worked with a complex client-server architecture where we have to get to know the frontend code implemented in html and JavaScript as well as the backend code in Elixir. Additionally, we had a lot of problems with the database connection and management through Ecto with the changesets and so on where we didn’t really get the hang of it until now. We therefore spent a lot of time understanding the architecture and structure of the Phoenix framework through tutorials and example projects. After some time, we finally understood how the frontend and backend interact with each other and where we can put our code so it makes sense and would work in the end. All this led to having our focus more on getting to know Phoenix than to implement and expand the more complex features we wanted to do in the beginning.

With more time and experience the implementation for sending images could successfully be developed in the future. Since the handling and storing of images is quite different from sending only text messages, it maybe would have been wise to implement a different, less complicated feature like authorization, deleting and modifying messages or similar.

A further step to optimize our solution would be to dockerize the application and make it *really* distributed. We did look into Docker and docker images and even tried to dockerize the application in the end but couldn’t get it do work also due to complications with database access from inside a container.

Bibliography

Chapter State-of-the-art

https://b-nova.com/home/content/phoenix-framework-the-killer-app-from-elixir visited last on 20.12.22

Chapter Presentation of the solution

https://github.com/dwyl/phoenix-chat-example visited last on 22.12.22

https://hexdocs.pm/phoenix/channels.html visited last on 22.12.22

https://info.codecast.io/blog/how-to-use-phoenix-channels visited last on 22.12.22

https://ludu.co/course/discover-elixir-phoenix/elixir-intro-installation/ visited last on 22.12.22

How (and when!) to store images in your database with Elixir & Phoenix | by Paul Fedory | Medium visited last on 20.12.22

File Uploads – Phoenix v1.3.0-rc.1 (hexdocs.pm) visited last on 20.12.22

Appendix

We host the Ocean application on a public GitHub repository where we also explain how to clone, setup and run the app. Please refer to https://github.com/damiankobler/Ocean.git in order to see all of the source code.

To see the successful start und running of the application we made a short video that opens the app in two browser windows and shows the functionalities and features we implemented like sending messages in multiple chatrooms and showing ocean animals. Please refer to https://youtu.be/mSrQhnMA3wY in order to access the video.