Konzepte moderner Programmiersprachen (KMPS)

Praktikum:
Phoenix LiveView

Wintersemester 2022/2023

Prof. Dr. Andreas Claßen
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

ChangeLog

2022-12-20: Pfade korrigiert (Änderungen in **blau**)

Hinweis zum Testatsystem

Für den "Prof. Faßbender" Teil sind im Testatsystem P1 - P3 vorgesehen.

Dieses Praktikum hier wird im Testatsystem als Praktikum Nr. 5 verbucht werden.

Es wird insgesamt nur diese zwei Praktika im „Claßen“ Teil geben…

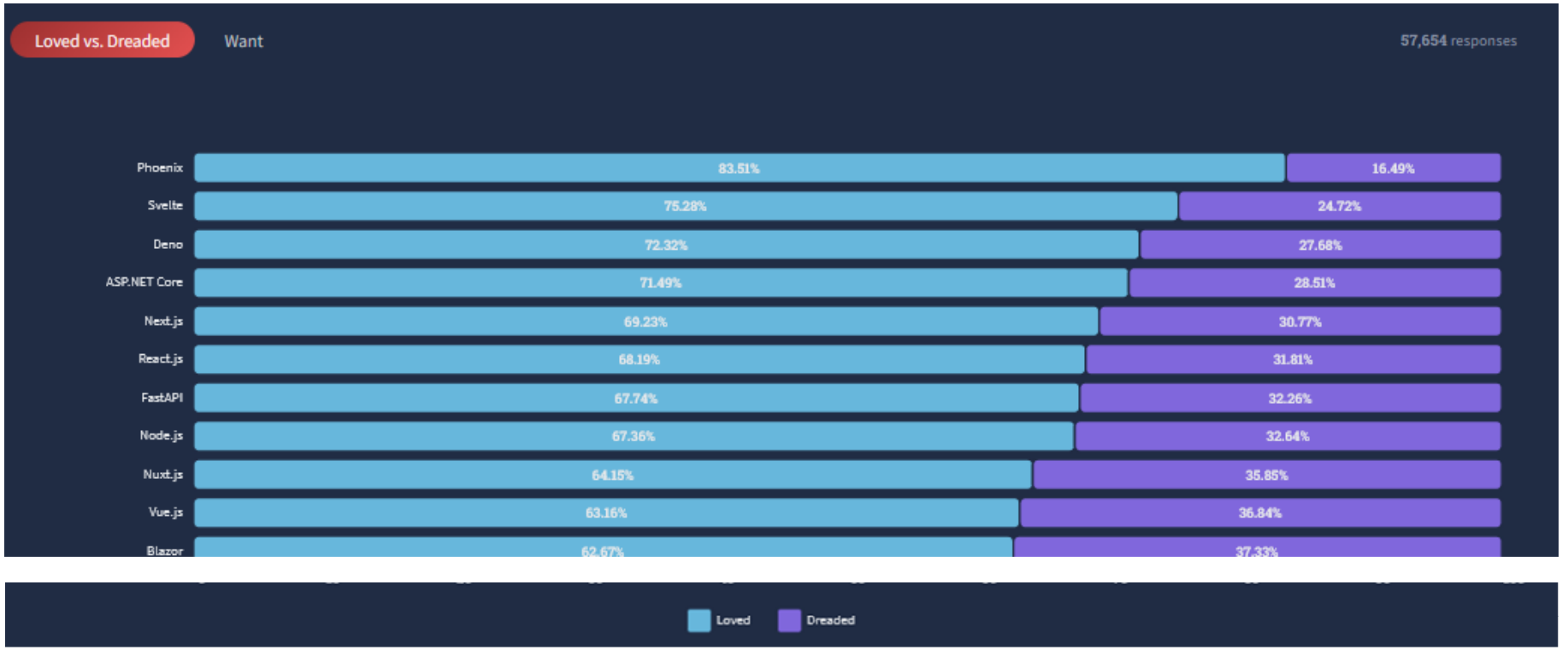
Elixir: Phoenix Web Framework

http://www.phoenixframework.org/
https://github.com/phoenixframework/phoenix

Stack Overflow Developer Survey 2022: Phoenix an Position 1 der Web Frameworks

Phoenix Framework

Web frameworks and technologies: Most loved, dreaded, and wanted



Quelle: https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted

Aufgaben dieses Praktikums

Aufgabe 1:
Erstellen bzw. erweitern Sie schrittweise gemäß den Vorgaben eine „Counter" App im Phoenix Web Framework (Sprache Elixir).

Ziele:
Die für Aufgabe 2 wesentlichen Mechanismen von Phoenix und Phoenix LiveWire kennenlernen.
Wertschätzen, wie einfach eine App mit Live Update Fähigkeiten dort zu realisieren ist.

Aufgabe 2:
Eine eigene (ganz einfache) Chat App in Phoenix LiveWire programmieren.

Methodik der Ergebnisabgabe für dieses Praktikum

Erstellen Sie ***eine*** `.zip` Datei von *wenigen KB*, in der folgende Dateien enthalten sind:

- Die von ihnen erstellten bzw. geänderten Source Code Dateien der Schritte der „Counter" App gemäß den Vorgaben
- Die von ihnen erstellten bzw. geänderten Source Code Dateien ihrer „Chat" App

Laden Sie diese *eine* `.zip` Datei nach Ilias hoch (in die "Übung" zu diesem Praktikum).

Einzelabgaben*! Teamarbeit ist erlaubt in dem Sinne, dass Sie die "technischen Knackpunkte" der Lösungen diskutieren und die Lösungsidee gemeinsam entwerfen. Aber die Lösungen sollten dann eine individuelle Umsetzung dieser gemeinsam entwickelten Ideen sein. D.h. Lösungen dürfen in Ansätzen ähnlich sein, aber nicht identisch.*

(Optional) Nutzung einer VM

Vorschlag: Nutzen Sie für dieses Praktikum die Debian 10 Virtuelle Maschine für VirtualBox, die Sie unter folgendem Link herunterladen können:
https://fh-aachen.sciebo.de/s/voHfcjRsXAfYbfJ/download?path=%2F&files=ITS_Client_Debian10_AMD64_V03-1.ova

Im Menü *Datei* von VirtualBox: *Appliance importieren*, heruntergeladene Datei *ITS_Client_Debian10_AMD64_V03-1.ova* auswählen, auf *Weiter* klicken.
Dann auf *Importieren* klicken.

User: `itsadmin`
Passwort: `itsadmin`

User hat `sudo` Rechte, d.h. kann über `sudo …` Befehle mit Admin-Rechten ausführen. Admin-User `root` hat auch `itsadmin` als Passwort.

Installation Elixir, Phoenix

 # In der VM:

 # Install Erlang & Elixir ...

```
wget https://packages.erlang-solutions.com/erlang-solutions_2.0_all.deb
sudo dpkg -i erlang-solutions_2.0_all.deb
sudo apt-get --allow-releaseinfo-change update
sudo apt-get install -y esl-erlang elixir curl kate
```

 # Install Node.js ...

```
sudo bash -c "curl -fsSL https://deb.nodesource.com/setup_16.x | bash - "
sudo apt-get install -y nodejs gcc g++ make
```

 # Install Phoenix ...

```
sudo apt-get install -y inotify-tools
mix archive.install --force hex phx_new
```

 # „rebar" und „rebar3" mit dem Erlang 25 Compiler neu compilieren ...
 # Sonst gibt es ggfs Fehlermeldungen …

```
mix local.rebar  # ...zweimal mit "y" akzeptieren
```

Elixir

Funktional, syntaktisch eher an imperativen Sprachen orientiert („versteckt“ funktionalen Charakter, um es Programmierern einfacher zu machen, die imperative Sprachen gewohnt sind). Nutzt Erlang VM BEAM. Erlang Libraries (insbes. OTP) in Elixir nutzbar.

Extrem geeignet für Parallelisierung durch funktionale Natur und Aktorenmodell, wie auch bei Erlang. Parallelisierung auf einem Host und sogar über mehrere Hosts hinweg (Cluster von Erlang/Elixir Nodes).

Phoenix Web Framework

Basiert auf Elixir als Programmiersprache, daher syntaktisch eher an imperativen Sprachen orientiert („versteckt" funktionalen Charakter, um es Programmierern einfacher zu machen, die imperative Sprachen gewohnt sind). Soll attraktiv sein z.B. für Ruby Programmierer, die von *Ruby on Rails* kommen.

Deklarativer Ansatz sehr gut geeignet für web-basierte Systeme:
Mehr in Richtung *was* will man erreichen (wie sollen die resultierenden Seiten aussehen, was möchte man nutzen, z.B. Authentifizierung möchte ich nutzen, PubSub möchte ich nutzen etc.) statt nur „imperatives *wie*" (*wie* muss ich was in den Gesamt-Kontrollfluss einbauen, damit es nachher funktioniert).

Skaliert durch Elixir/Erlang Basis extrem gut auch für sehr performante Webserver-Cluster (falls extrem hohe Anfragelast), ohne dass zusätzliche Technologien dafür hinzugefügt werden müssten oder die SW anders (aufwändiger, komplexer, spezifisch) dafür programmiert werden müsste.

Client-Code (JavaScript Code) wird generiert, d.h. man bräuchte den Client gar nicht zu programmieren. *Kann man natürlich trotzdem machen bzw. erweitern, wenn man z.B. JS UI Frameworks im Client nutzen möchte.*

Phoenix LiveWire

… ist mittlerweile „eingebauter Teil“ des Phoenix Frameworks und erweitert dieses um die „Near Real-Time“ Update-Fähigkeiten (PubSub über WebSockets), ohne dass der Programmierer die dafür notwendige technologische Basis (z.B. WebSockets) selbst über eigenen Code erzeugen bzw. handhaben müsste.

Aufgabe 1
Schritt 1: App „Skelett“ anlegen lassen (scaffolding)

Lassen Sie in einem beliebigen Verzeichnis durch Phoenix eine neue App `counter` anlegen:

```
export BASE_DIR=$(pwd)

mix phx.new counter --install --no-ecto
# Option --no-ecto bedeutet "Web Backend ohne Datenbank-Anbindung"

cd counter
mkdir lib/counter_web/live
echo "# empty" > lib/counter_web/live/counter_live.ex
```

Aufgabe 1
Schritt 2: Phoenix Webserver starten

… mittels des folgenden Befehls:

```
cd $BASE_DIR/counter
mix phx.server
```

Der Webserver (und damit der Befehl) wird durchgehend laufen, bis er mittels `Ctrl-C` / `Strg-C` und anschließendem `a` (für `abort`) beendet wird.

Um weitere Befehle in die Kommandozeile eingeben zu können, öffnen Sie dort mittels `Ctrl-Shift-n` ein weiteres Befehlsfenster.

Öffnen Sie in der VM einen Firefox Browser und schauen Sie sich die URL `localhost:4000` an. Dort müsste die Webseite des Phoenix Frameworks zu sehen sein.

Aufgabe 1
Schritt 3: Aktuellen Stand sichern als „Stufe 0“

Sichern Sie die essentiellen Dateien:

```
cd $BASE_DIR

mkdir counter_0/

cp counter/lib/counter_web/router.ex counter_0/
cp counter/lib/counter_web/live/counter_live.ex counter_0/

cd $BASE_DIR/counter
```

Aufgabe 1
Schritt 4a: Routing für View `/counter`

Fügen Sie in Datei `counter/lib/counter_web/router.ex` unterhalb der Zeile …

```
get "/", PageController, :index
```

… folgende neue Zeile ein:

```
live "/counter", CounterLive
```

D.h. der View zur URL `/counter` wird vom "Live Controller" `CounterLive` gehandhabt.

```
scope "/", CounterWeb do
    pipe_through :browser
    get "/", PageController, :index
    live "/counter", CounterLive
end
```

Aufgabe 1
Schritt 4b: „Live Controller“ `CounterLive`

Fügen Sie in der Datei `counter/lib/counter_web/live/counter_live.ex` folgenden Dateiinhalt ein:

```
defmodule CounterWeb.CounterLive do
    use CounterWeb, :live_view

    def mount(params, session, socket) do
        {:ok, assign(socket, :counter_value, 0)}
    end

    def render(assigns) do
        ~H"""
            <h1>Live Counter</h1>
            <div id="counter">
                Counter value: <%= @counter_value %>
            </div>
        """
    end
end
```

Aufgabe 1
Schritt 4b: „Live Controller“ `CounterLive`

Zur Erläuterung:

Beim Aufbau des Web-Clients (`mount`) wird eine Variable `counter_value` bereitgestellt und mit dem Wert `0` initialisiert.

`counter_value` muss mittels Doppelpunkt „ge-quoted“ werden (`:counter_value`), da an dieser Stelle der Name der Variable referenziert werden soll und nicht der Wert der Variable, wie das sonst automatisch gemacht wird, wenn Variablennamen „an Wert-Positionen“ im Programmcode stehen …

Bei imperativen Programmiersprachen werden Variablennamen auf der linken Seite der Zuweisung automatisch gequoted, während auf der rechten Seite durch den Wert ersetzt wird, z.B. bei `i = i+1;` Aber auch nicht immer, z.B. nicht bei `*int_ptr = *int_ptr + 1;` (lvalue versus rvalue).

```
defmodule CounterWeb.CounterLive do
    ...

    def mount(params, session, socket) do
        {:ok, assign(socket, :counter_value, 0)}
    end

    ...
```

Aufgabe 1
Schritt 4b: „Live Controller“ `CounterLive`

Zur Erläuterung:

Beim Rendern der Webseite wird der angegebene HTML Code in den Body der Seite eingefügt, wobei `<% … %>` Bereiche als Elixir Code ausgeführt werden und hier im konkreten Fall der aktuelle Wert der Variable `counter_value` ausgelesen wird.
`@counter_value` ist eine abkürzende Schreibweise (Makro) innerhalb von Templates für den Zugriff auf Einträge in `assigns`, also auf Variablen. Sonst müsste man `Map.get(assigns, :counter_value)` schreiben …

```
defmodule CounterWeb.CounterLive do
    ...

    def render(assigns) do
        ~H"""
            <h1>Live Counter</h1>
            <div id="counter">
                Counter value: <%= @counter_value %>
            </div>
        """
    end
```

Aufgabe 1
Schritt 5: Aktuellen Stand sichern als „Stufe 1“

Schauen Sie sich den aktuelle Stand im Web Browser an unter `http://localhost:4000/counter`.
Durch Live Reload ist kein Neustart des Webservers o.ä. nötig.

Sichern Sie die essentiellen Dateien:

```
cd $BASE_DIR

mkdir counter_1/

cp counter/lib/counter_web/router.ex counter_1/
cp counter/lib/counter_web/live/counter_live.ex counter_1/

cd $BASE_DIR/counter
```

Aufgabe 1, Schritt 6a: Buttons im „Live Controller“
CounterLive

Erweitern Sie in der Datei `counter/lib/counter_web/live/counter_live.ex` die Funktion `render()` wie folgt. Dies fügt der HTML Seite Buttons hinzu, die aber noch „ohne Wirkung“ sind.

```
defmodule CounterWeb.CounterLive do
    ...
    def render(assigns) do
        ~H"""
            <h1>Live Counter</h1>
            <div id="counter">
                Counter value: <%= @counter_value %>
            </div>
            <button phx-click="add-1"> +1 </button>
            <button phx-click="add-5"> +5 </button>
            <button phx-click="add-10"> +10 </button>
        """
    end
end
```

Aufgabe 1, Schritt 6b: „Button-Handler“ im „Live Controller“ `CounterLive`

Erweitern Sie die Datei `counter/lib/counter_web/live/counter_live.ex` um Handler-Funktionen für die Buttons. `assigns` sind die „Variablenzuweisungen“.

```
defmodule CounterWeb.CounterLive do
    ...
    def render(assigns) do
        ...
    end
    def handle_event("add-1", _, socket) do
        counter_value = socket.assigns.counter_value + 1
        {:noreply, assign(socket, :counter_value, counter_value)}
    end
    def handle_event("add-5", _, socket) do
        counter_value = socket.assigns.counter_value + 5
        {:noreply, assign(socket, :counter_value, counter_value)}
    end
    def handle_event("add-10", _, socket) do
        counter_value = socket.assigns.counter_value + 10
        {:noreply, assign(socket, :counter_value, counter_value)}
    end
end
```

Name der Variable (linke Seite der Wertzuweisung)

Wert der Variable (rechte Seite der Wertzuweisung)

Aufgabe 1
Schritt 7: Aktuellen Stand sichern als „Stufe 2“

Schauen Sie sich den aktuelle Stand im Web Browser an.
Durch Live Reload ist kein Neustart des Webservers o.ä. nötig.

Sichern Sie die essentiellen Dateien:

```
cd $BASE_DIR

mkdir counter_2/

# router.ex ist unveraendert ...
cp counter/lib/counter_web/router.ex counter_2/

cp counter/lib/counter_web/live/counter_live.ex counter_2/

cd $BASE_DIR/counter
```

Aufgabe 1, Schritt 8a: Subscribe auf Änderungen im „Live Controller“ `CounterLive`

Erweitern Sie die Datei `counter/lib/counter_web/live/counter_live.ex` um die Subscription auf das Topic `counter_increase`. Dadurch können mehrere Clients Updates „in Real-Time“ empfangen und umsetzen. Die Subscription wird „beim Aufbau des Clients“ (`mount`) angemeldet.

```
defmodule CounterWeb.CounterLive do
    use CounterWeb, :live_view

    @topic "counter_increase"

    def mount(params, session, socket) do
        CounterWeb.Endpoint.subscribe(@topic)
        {:ok, assign(socket, :counter_value, 0)}
    end

    ...
end
```

Aufgabe 1, Schritt 8b: Handler für per Subscribe empfangene Änderungen im „Live Controller“ `CounterLive`

Erweitern Sie die Datei `counter/lib/counter_web/live/counter_live.ex` um eine `handle_info()` Funktion zur Reaktion auf per Subscription empfangene Nachrichten zum Topic.

```
defmodule CounterWeb.CounterLive do
    ...

    def handle_info( %{topic: @topic, payload: new_counter_value}, socket ) do
        {:noreply, assign(socket, :counter_value, new_counter_value)}
    end

end
```

Aufgabe 1, Schritt 8c: Publish von Änderungen im „Live Controller“ `CounterLive`

Erweitern Sie in der Datei `counter/lib/counter_web/live/counter_live.ex` die `handle_event()` Funktionen um ein broadcast-publish der Counter-Änderung.

```
defmodule CounterWeb.CounterLive do
    ...

 def handle_event("add-1", _, socket) do
       counter_value = socket.assigns.counter_value + 1
       CounterWeb.Endpoint.broadcast_from(self(), @topic,
                                          "counter_increase_event",
                                          counter_value)
       {:noreply, assign(socket, :counter_value, counter_value)}
    end

    # ... identische Zeile auch in handle_event() für "add-5", "add-10"
end
```

Aufgabe 1
Schritt 9: Finalen Stand sichern als „Stufe 3“

Schauen Sie sich den aktuelle Stand im Web Browser an.
Durch Live Reload ist kein Neustart des Webservers o.ä. nötig.

Öffnen Sie aus dem Web Browser heraus ein oder mehrere neue „private Fenster“ und arbeiten Sie mit diesen mehreren Clients.
Der Client holt beim `mount` nicht den aktuellen Stand der Counter-Variable. Daher bekommen Clients erst beim ersten Counter-Update den aktuellen Counter-Wert mitgeteilt… Dies wurde so realisiert, um den Code einfach zu halten …
"Private Fenster" bedeuten, dass keine Daten geteilt werden. Damit ist sichergestellt, dass der Datenaustausch über den Server stattfinden muss und nicht im Wegbrowser zwischen den Fenstern.

Sichern Sie die essentiellen Dateien:

```
cd $BASE_DIR
mkdir counter_3/
# router.ex ist unveraendert ...
cp counter/lib/counter_web/router.ex counter_3/
cp counter/lib/counter_web/live/counter_live.ex counter_3/
cd $BASE_DIR
```

Aufgabe 1
Schritt 10: Abgabe für Aufgabe 1

Die Inhalte der Verzeichnisse `counter_0`, `counter_1`, `counter_2`, `counter_3` sollen in ihrer *einen* finalen `zip` Abgabedatei enthalten sein.

Sie sollen **nicht den gesamten Code der `counter` Applikation abgeben**, das wären dann ca 26 MB, sondern nur die „Varianten" der beiden Dateien `router.ex` und `counter_live.ex`, also wenige KB.

*Halten Sie sich bitte daran, ich betrachte das als **Intelligenztest** und sie wollen nach dem Studium ja fürs Denken bezahlt werden … ;-)*

Aufgabe 2: Eine einfache Chat App

… in Elixir, Phoenix und LiveView, basierend auf den Konzepten (und ggfs. dem Code) aus Aufgabe 1.

D.h. die App soll Multi-Client fähig sein und neue Chat Nachrichten „in Real-Time" an die anderen Clients propagieren.

Generelles Prinzip: nur „ganz einfache Realisierung gefragt". Also z.B. …
… keine DB im Backend, kein Login, nur „simpler einzeiliger Text" ist ok (keine Sonderzeichen / Zeilenumbrüche etc im Text o.ä.), keine „links-rechts Positionierung" betreffend Sender-Empfänger, nur ein einziger gemeinsamer Chat, in den alle Clients schreiben und den alle Clients lesen/sehen.

Aufgabe 2: Chat App - Randbedingungen

Jeder Client hat ein einzeiliges Textfeld, in welches der User den Namen dieses Clients einträgt. *Ob diese Eintragung durch einen Button im UI bestätigt werden muss oder nicht bleibt ihnen überlassen …*

In ein weiteres (einzeilig o.k.) Textfeld kann der User die nächste Textnachricht dieses Clients eintragen und durch einen SEND Button abschicken.

Jeder Client zeigt die ab seinem Beitritt gesendeten Nachrichten an sowie zu jeder Nachricht den Namen des Clients, der diese Nachricht gesendet hat (Datum/Uhrzeit des Sendens o.ä. braucht nicht gespeichert bzw. angezeigt zu werden). *Nur die Nachrichten ab dem Beitritt, nicht die Historie der Kommunikation, die ggfs. schon vor dem Beitritt stattgefunden hat.*

Es ist o.k., wenn das Backend bei jedem Update die komplette Historie aller Nachrichten an alle Clients sendet. Es muss nicht nur die neueste versendete Nachricht gesendet werden (keine Minimierung des Netzwerk-Traffics gefordert).

Das UI muss nicht „schön" sein. Es geht „nur um Basisfunktionalität".

Aufgabe 2: App „Skelett“ anlegen lassen (scaffolding)

Lassen Sie von Phoenix eine neue App `chat` anlegen:

```
cd $BASE_DIR

mix phx.new chat --install --no-ecto
# Option --no-ecto bedeutet "Web Backend ohne Datenbankanbindung"

cd chat
mkdir lib/chat_web/live
echo "# empty" > lib/chat_web/live/chat_live.ex

cd $BASE_DIR/chat
mix phx.server
firefox localhost:4000/chat &
# … solange Datei lib/chat_web/router.ex nicht angepasst wird:
#   Erst einmal Fehler "no route found for GET /chat"

# im Editor bearbeiten: folgende zwei Dateien ...
# lib/chat_web/router.ex
# lib/chat_web/live/chat_live.ex
```

Abgabe für Aufgabe 2

Packen Sie in ihre *eine* `.zip` Abgabedatei in ein eigenes Verzeichnis `chat_abgabe` nur die Quellcode-Dateien hinein, die Sie geschrieben bzw. geändert haben!

*Also wiederum **nicht** alle Dateien des Phoenix Projekts für die Chat App!*
Fortsetzung des Intelligenztests…

*Nur **Einzelabgaben** erlaubt!*
Kopierte Lösungen werden nicht akzeptiert (und es wird kein Unterschied gemacht, wer Originalautor und wer „Empfänger" des Kopiervorgangs war…)!
Machen Sie ihre Lösung auch nicht in einem Public Repository verfügbar.
(Wenn dann jemand von dort kopiert, ist das auch eine kopierte Lösung, selbst wenn Sie sich des Kopiervorgangs gar nicht bewusst waren. Ist schon vorgekommen… Der „public" Status war dann ja ihre Entscheidung…)