Chat Zumbizade™ com ABAP Channels e UI5

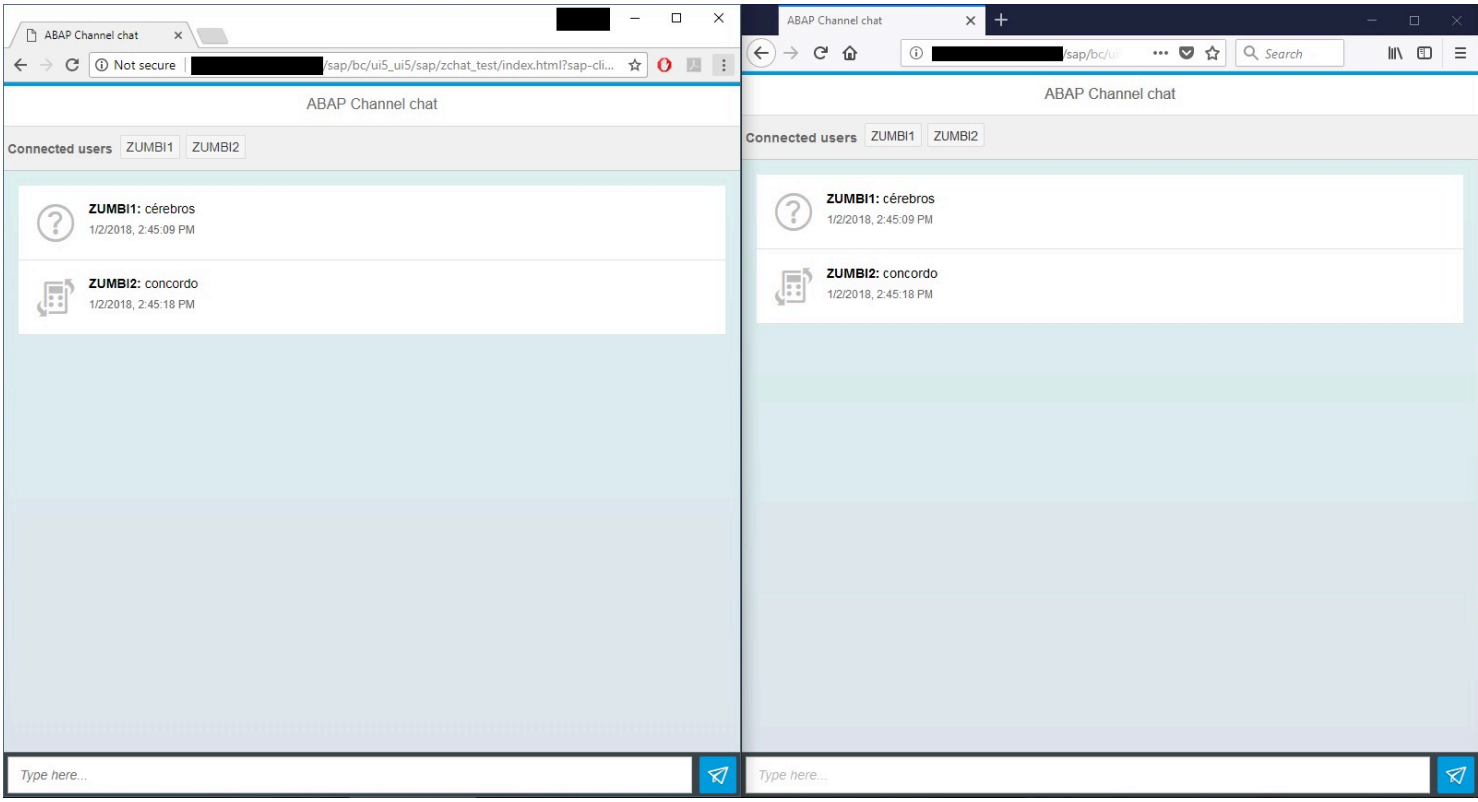
03/01/2018 10:00

Fala zumbizada!

Saindo da tumba após eras de profundo adormecimento, cá estou eu de volta à linha de frente zumbi pra despejar um pouco mais de conhecimento para nós, seres meio-viventes. Dessa vez, vim pra mostrar um exemplo lúdico de algo que comentei no último episódio do nosso cast (que sim, saiu ano passado, porém não se desesperem pois novos episódios estão a caminho- sim, com S): uma aplicaçãozinha *chat* feita com ABAP Channel e UI5.

Pra quem não sabe, **ABAP Channel** é a implementação ABAP de *web sockets*. Se você não entende patavinas deste assunto, recomendo fortemente a leitura do *post* da SAP Community (é assim que chama agora né?) que eu indiquei no The Walking Dev #008 (eis o link outra vez). *Web sockets* são coisas bem interessantes, que tem ótimas possibilidades de aplicação dentro do SAP, como por exemplo para a criação de *dashboards real-time*, clientes de *workflow*, ferramentas de cooperação, etc.

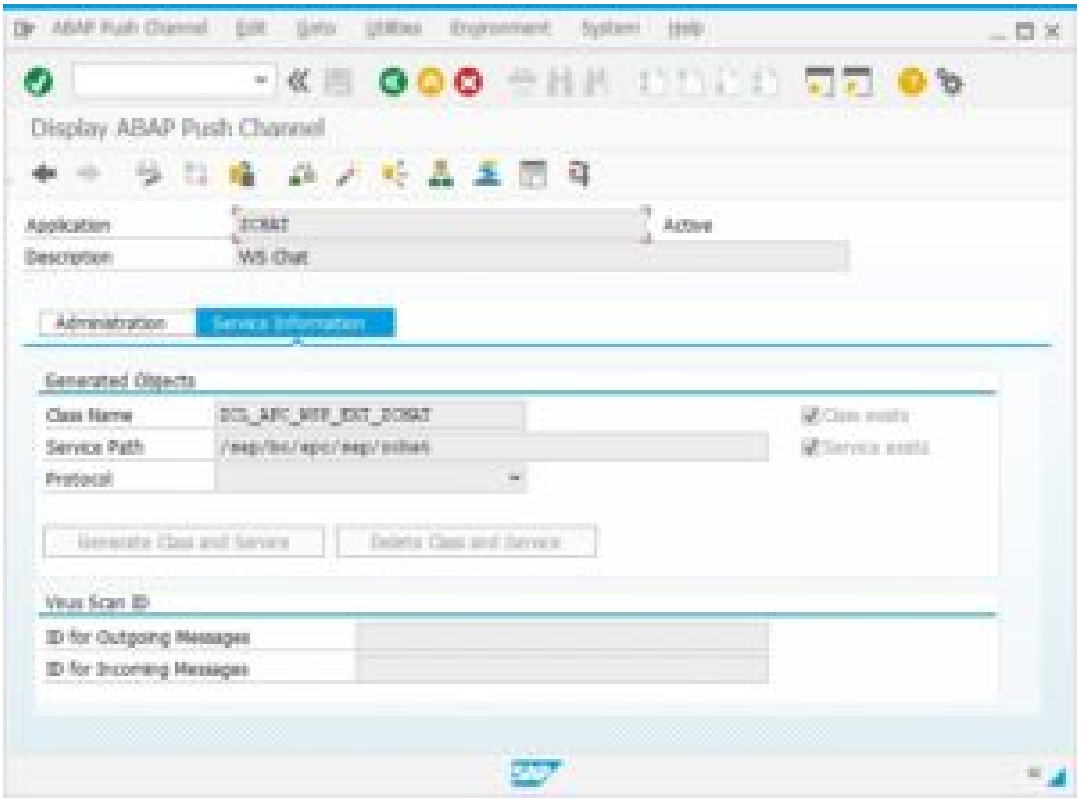
A ideia desse exemplo é construir um *chat* simples com uma sala única pra todos os usuários logados em um mesmo sistema. Cada usuário entra com o próprio ID na sala e um avatar escolhido aleatoriamente dentre os ícones disponíveis no UI5. As mensagens são exibidas com o *timestamp* gerado pelo servidor.



Maurício fala reservadamente para todos: oi e vc

Vamos aos objetos:

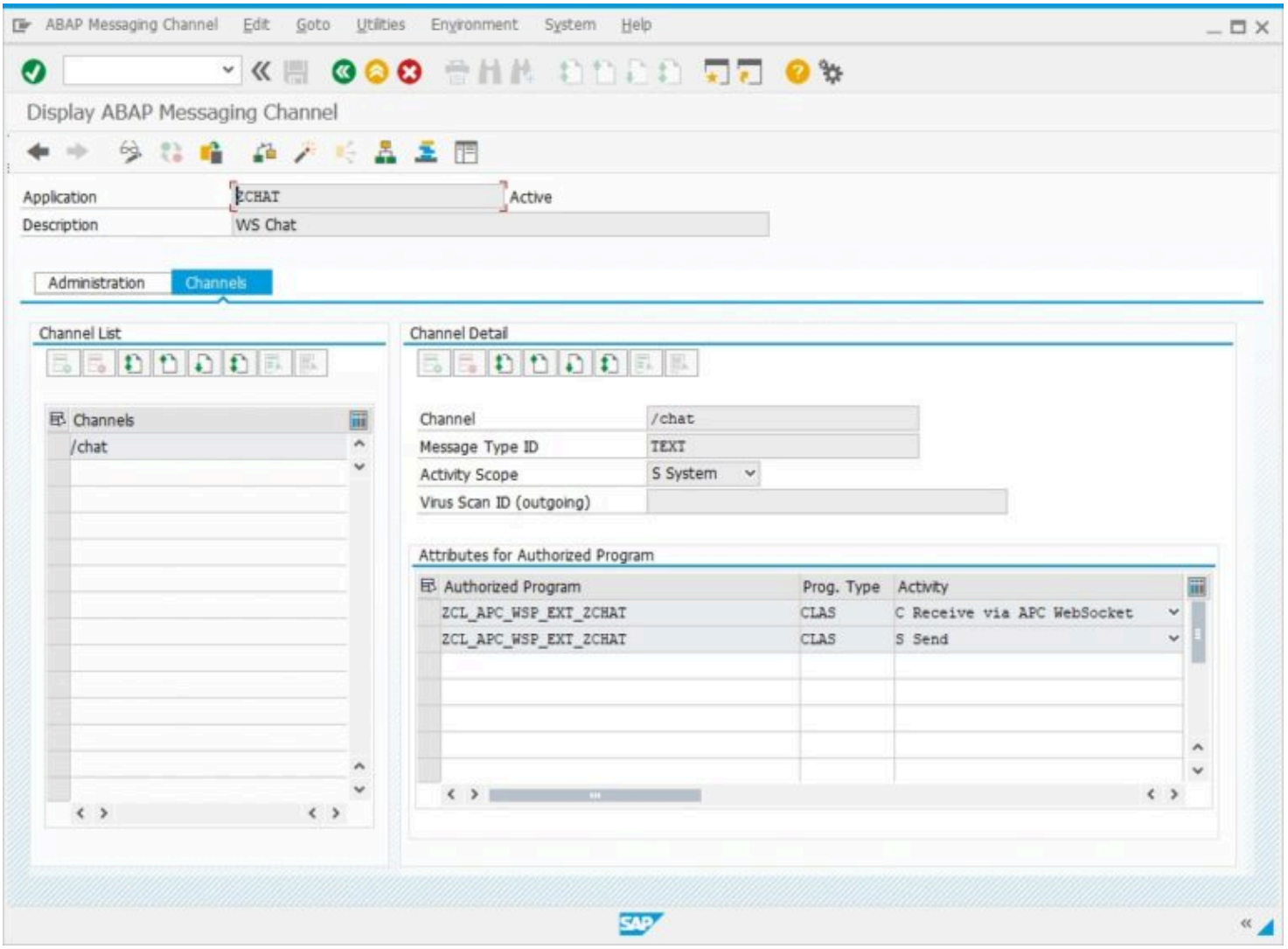
Push channel



Esse nome de classe escroto foi gerado automaticamente blz?

Aqui não tem nada de complicado, basta dar um nome pra sua aplicação e uma descrição e clicar ali em **Generate Class and Service**. Salvar e ativar, como sempre.

Message channel



Dei o mesmo nome pro message channel por motivos de preguiça

O *message channel* também é bem simples: só tem um canal,/chat, que vai ser do tipo TEXT e com escopo S (*System*). Aí adicionamos a classe gerada pro *push channel* na lista de programas autorizados para as atividades C (receber dados por *push channel*) e S (enviar). Novamente, salvar e ativar.

Implementação da classe

Private section

```
private section.

  types: begin of LTY_CONNECTED_USER,
           id TYPE syuname,
           avatar TYPE i,
         end of LTY_CONNECTED_USER.

  methods SEND_MESSAGE
    importing
      !IM_EXTENSION_ID type AMC_CHANNEL_EXTENSION_ID
      !IM_ROOT_ID type ABAP_TRANS_SRCNAME
      !IM_DATA type ANY .
```

Aqui temos um tipo pra criar uma tabela de usuários conectados no *chat*, e a declaração do método que vai ser usado para enviar mensagens.

IF_APC_WSP_EXTENSION~ON_START

```
METHOD if_apc_wsp_extension~on_start.

  DATA lt_users TYPE SORTED TABLE OF lty_connected_user WITH UNIQUE KEY id.

  TRY.
      i_context->get_binding_manager( )->bind_amc_message_consumer(
          i_application_id       = 'ZCHAT'
```

```
          i_channel_id           = '/chat'
          i_channel_extension_id = '/users'
      ).
      i_context->get_binding_manager( )->bind_amc_message_consumer(
          i_application_id       = 'ZCHAT'
          i_channel_id           = '/chat'
          i_channel_extension_id = '/messages'
      ).
      DATA(lv_avatar) = i_context->get_initial_request( )->get_form_field( 'avatar' ).
    CATCH cx_apc_error INTO DATA(lx_apc_error).
      MESSAGE lx_apc_error->get_text( ) TYPE 'E'.
  ENDTRY.

  IMPORT users = lt_users FROM DATABASE indx(zc) ID 'ZCHAT'.
  READ TABLE lt_users ASSIGNING FIELD-SYMBOL(<user>) WITH KEY id = sy-uname.
  IF sy-subrc = 0.
    <user>-avatar = lv_avatar.
  ELSE.
    INSERT VALUE lty_connected_user( id = sy-uname avatar = lv_avatar ) INTO lt_users INDEX sy-tabix.
  ENDIF.
  EXPORT users = lt_users TO DATABASE indx(zc) ID 'ZCHAT'.

  me->send_message(
      im_extension_id = '/users'
      im_root_id      = 'connectedUsers'
      im_data         = lt_users
  ).

ENDMETHOD.
```

Esse método é chamado assim que a conexão do *web socket* é iniciada.

Na primeira parte, fazemos o *binding* do *push channel* com o *message channel*, criando duas extensões diferentes: /users, onde vão trafegar as mensagens sobre conexão e desconexão de usuários, e /messages, onde vão trafegar as mensagens do *chat* em si. Esse recurso das extensões é muito bacana, porque permite segmentar as mensagens sem ter que criar canais diferentes na mesma aplicação.

Quando a conexão é criada, a aplicação UI5 vai chamar o serviço passando um parâmetro de URL chamado avatar, que vai conter o ID do avatar gerado aleatoriamente pelo *frontend* (porque é lá que fica a coleção de ícones). Juntamos esse valor com o nome do usuário e gravamos a tabela toda num cluster na INDX (tomando cuidado pra ver se o usuário já está logado). Achei mais simples usar EXPORT ... TO DATABASE nesse caso, mas num caso normal talvez seria melhor usar *shared memory*.

Por fim, enviamos na extensão /users uma mensagem com a tabela atualizada de usuários conectados.

IF_APC_WSP_EXTENSION~ON_MESSAGE

```
METHOD if_apc_wsp_extension~on_message.

  DATA: BEGIN OF ls_message,
          text      TYPE string,
          sender    TYPE string,
          timestamp TYPE string,
        END OF ls_message.

  TRY.
      ls_message-text = i_message->get_text( ).
    CATCH cx_apc_error INTO DATA(lx_apc_error).
      MESSAGE lx_apc_error->get_text( ) TYPE 'E'.
  ENDTRY.

  ls_message-sender = sy-uname.
  GET TIME STAMP FIELD DATA(lv_timestamp).
  ls_message-timestamp = |{ lv_timestamp TIMESTAMP = ISO TIMEZONE = sy-zonlo }|.

  me->send_message(
      im_extension_id = '/messages'
      im_root_id      = 'message'
      im_data         = ls_message
  ).

ENDMETHOD.
```

Esse método é chamado toda vez que o *push channel* recebe uma mensagem – no nosso caso, é sempre uma mensagem do *chat*. Portanto, é uma implementação bem simples: pegamos o texto, marcamos com o ID do usuário e o *timestamp* formatado como ISO no fuso local, e retornamos na extensão /messages.

IF_APC_WSP_EXTENSION~ON_CLOSE

```
METHOD if_apc_wsp_extension~on_close.

  DATA lt_users TYPE SORTED TABLE OF lty_connected_user WITH UNIQUE KEY id.

  IMPORT users = lt_users FROM DATABASE indx(zc) ID 'ZCHAT'.
  DELETE lt_users WHERE id = sy-uname.
  IF lines( lt_users ) = 0.
```

```
    DELETE FROM DATABASE indx(zc) ID 'ZCHAT'.
  ELSE.
    EXPORT users = lt_users TO DATABASE indx(zc) ID 'ZCHAT'.
  ENDIF.

  me->send_message(
      im_extension_id = '/users'
      im_root_id      = 'connectedUsers'
      im_data         = lt_users
  ).

ENDMETHOD.
```

Esse método é chamado ao encerrar a conexão. Aqui, precisamos notificar todos que houve uma desconexão, então verificamos a tabela de usuários e removemos o ID que desconectou, tomando o cuidado de zerar a tabela do *cluster* na INDX quando não houver mais ninguém no *chat*. A tabela final é enviada na extensão /users.

SEND_MESSAGE

```
METHOD send_message.

  DATA: lo_producer TYPE REF TO if_amc_message_producer_text,
        lv_key      TYPE        string.

  DATA(lt_source) = VALUE abap_trans_srcbind_tab( ( name = im_root_id value = REF #( im_data ) ) ).
  DATA(lo_writer) = cl_sxml_string_writer=>create( type = if_sxml=>co_xt_json ).
  CALL TRANSFORMATION id SOURCE (lt_source) RESULT XML lo_writer.
  DATA(lv_message) = cl_abap_codepage=>convert_from( lo_writer->get_output( ) ).
  DO.
    CLEAR lv_key.
    lv_key = match( val = lv_message regex = '"[0-9_\-]*[A-Z]+[A-Z0-9_\-]*":' ).
    IF lv_key IS INITIAL.
      EXIT.
    ELSE.
      REPLACE ALL OCCURRENCES OF lv_key IN lv_message WITH to_lower( lv_key ).
    ENDIF.
  ENDDO.

  TRY.
      lo_producer ?= cl_amc_channel_manager=>create_message_producer(
          i_application_id       = 'ZCHAT'
          i_channel_id           = '/chat'
          i_channel_extension_id = im_extension_id
      ).
      lo_producer->send( lv_message ).
    CATCH cx_amc_error INTO DATA(lx_amc_error).
      MESSAGE lx_amc_error->get_text( ) TYPE 'E'.
  ENDTRY.

ENDMETHOD.
```

Esse é um método utilitário para enviar mensagens pelo *message channel*. Basicamente, pegamos os dados que precisam ser enviados (em qualquer formato que estejam: variável simples, estrutura, tabela, etc.), chamamos o CALL TRANSFORMATION id para gerar uma saída JSON através da CL_SXML_STRING_WRITER, e através de uma expressão regular deixamos todos os identificadores do JSON em minúsculas.

Por fim, criamos um *message producer* com a extensão necessária e enviamos a mensagem.

UI5

Chat.view.xml

```
<mvc:View controllerName="com.abapzombie.chat.controller.Chat" xmlns:html="http://www.w3.org/1999/xhtml" xmlns:mvc="sap.ui.core.mvc" displayB
  <App>
    <pages>
      <Page id="chatWindow" class="sapUiContentPadding" title="{i18n>title}">
        <subHeader>
          <Toolbar>
            <Label design="Bold" text="{i18n>usersLabel}"/>
            <Tokenizer tokens="{ path: 'user>/connectedUsers', sorter: 'id' }">
              <Token key="{user>id}" text="{user>id}" editable="false"/>
            </Tokenizer>
          </Toolbar>
        </subHeader>
        <footer>
          <Toolbar>
            <Input id="message" placeholder="{i18n>messagePlaceholder}" change="onSend"/>
            <Button type="Emphasized" icon="sap-icon://paper-plane" press="onSend"/>
          </Toolbar>
        </footer>
      </Page>
    </pages>
  </App>
</mvc:View>
```

Chat.controller.js

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/ui/model/json/JSONModel",
  "sap/m/FeedListItem"
], function(Controller, JSONModel, FeedListItem) {
  "use strict";

  return Controller.extend("com.abapzombie.chat.controller.Chat", {

    onInit: function() {
      this.getView().setModel(new JSONModel({
        connectedUsers: []
      }), "user");
      this._setChatWindowScroll();
      this._setMessageInputKeydown();
      this._setWebSocket();
    },

    onSend: function() {
      var oMessageInput = this.byId("message"),
          sText = oMessageInput.getValue();
      if (sText) {
        this._wsChat.send(sText);
      }
      oMessageInput.setValue("");
    },

    onExit: function() {
      this._wsChat.close();
    },

    _getAvatar: function(sSender) {
      var aConnectedUsers = this.getView().getModel("user").getProperty("/connectedUsers"),
          aSender = jQuery.grep(aConnectedUsers, function(user) {
            return user.id === sSender;
          });
      if (aSender.length === 1) {
        return "sap-icon://" + sap.ui.core.IconPool.getIconNames()[aSender[0].avatar];
      }
    },

    _setMessageInputKeydown: function() {
      this.byId("message").attachBrowserEvent("keydown", function(oEvent) {
        if (oEvent.which === 9) {
          oEvent.preventDefault();
        }
      });
    },

    _setChatWindowScroll: function() {
      var oChatWindow = this.byId("chatWindow");
      oChatWindow.addEventDelegate({
        onAfterRendering: function() {
          var aContent = oChatWindow.getContent();
          if (aContent.length > 0) {
            var $lastItem = jQuery.sap.byId(aContent.pop().sId);
            oChatWindow.scrollTo($lastItem.position().top + $lastItem.outerHeight());
          }
        }
      }, this);
    },

    _setWebSocket: function() {
      var sWSURL = "ws://" + window.location.host + "/sap/bc/apc/sap/zchat?" + jQuery.param({
            avatar: Math.floor(Math.random() * sap.ui.core.IconPool.getIconNames().length)
          }),
          oUserModel = this.getView().getModel("user"),
          oChatWindow = this.byId("chatWindow"),
          that = this;
      this._wsChat = new WebSocket(sWSURL);
      this._wsChat.addEventListener("message", function(oEvent) {
        var oData = JSON.parse(oEvent.data);
        if (oData.connectedUsers) {
          oUserModel.setProperty("/connectedUsers", oData.connectedUsers);
        }
        if (oData.message) {
          oChatWindow.addContent(new FeedListItem({
            icon: that._getAvatar(oData.message.sender),
            iconActive: false,
            sender: oData.message.sender,
            senderActive: false,
            text: oData.message.text,
            timestamp: (new Date(oData.message.timestamp)).toLocaleString()
          }));
        }
      });
    }

  });
```

```
});
```

Tanto a *view* quanto o *controller* são bem simples.

A *view* consiste em uma página com o conteúdo principal vazio, que é onde vão aparecer as mensagens assim que elas forem recebidas. O conteúdo fixo são duas *toolbars*, uma no topo com um *tokenizer* para exibir os usuários conectados e outra na parte de baixo com o *input* e o botão para enviar mensagens.

O *controller* tem as seguintes atribuições:

- Criar um *JSONModel* para alimentar o *tokenizer* (feito diretamente no onInit);
- Definir o *scroll* automático da página, para quando novas mensagens forem adicionadas (método _setChatWindowScroll);
- Impedir o uso de [Tab] no *input*, para que somente o [Enter] envie mensagens (método _setMessageInputKeydown);
- Configurar o *web socket* (finalmente! pelo método _setWebSocket), onde:
    - Define-se um avatar aleatório, pelo índice do *IconPool*;
    - Abre-se a conexão com o *web socket*;
    - Define-se um *event handler* para quando chegarem mensagens:
        - Se houver o identificador connectedUsers, definir o conteúdo dentro do *JSONModel*;
        - Se houver o identificador message, criar um novo *FeedListItem* com os dados da mensagem;
- Enviar a mensagem (método onSend);
- E fechar a conexão com o *web socket* (diretamente no método onExit).

E é basicamente isso. Agora, caro amigo zumbi, vc tem um chat pra conversar com os seus outros amiguinhos zumbis no projeto.

Nas versões mais novas do UI5 já existe a classe sap.ui.core.ws.WebSocket, então se você estiver surfando na crista da onda já dá pra usar ela em vez da classe JS pura que eu usei aqui.

Dúvidas, sugestões ou comentários? Só postar que a gente se fala.

PS.: ~~Nada disso está no Github por motivos de preguiça 😬~~ Mentira! Tá lá no Github sim 🙂

Comentários

Anderson — *21/06/2019 16:40*

tem que colocar alguma coisa no neo-app.json? existe alguma configuração adicional no destination la no SCP ? i need it!!!

> **Leo Schmidt** — *21/06/2019 17:18*
>
> Fala Anderson, blz?
>
> Pelo que eu entendi, vc tá desenvolvendo no SCP Trial e usando uma destination via Cloud Connector.
>
> Nesse caso, teu neo-app.json tem que apontar os recursos pra destination, senão vc vai usar os recursos locais do SCP. Algo do tipo:
>
> ```
> {
> "path": "/destinations/northwind",
> "target": {
> "type": "destination", //tem que indicar essa propriedade aqui
> "name": "northwind" //aqui vc usa o nome da sua destination
> },
> "description": "Northwind OData Service"
> }
> ```
>
> Não tentei ainda usar uma configuração desse tipo nesse projeto, não sei se funcionaria por causa do caminho do ABAP Channel (`ws://`). Quando tiver um tempo, posso tentar aqui e ver se funciona.
>
> Abraço!
>
> > **Anderson** — *21/06/2019 17:37*
> >
> > Grande Leo!! isso aí... esse é o cenário... o desafio é justamente esse, passar pelo cloud connector do SCP .. Pensei até que seu exemplo fosse desse jeito... Ficarei grato se conseguir dessa forma e compartilhar aqui. Encontrei uma nota onde diz que o CloudConnector tem que ser na versão 2.12 (a ultima) ...
> > fiz algo pareciso com a definição do destination que vc mencionou, porém sem sucesso... Ainda estou aqui na luta. De qquer forma se eu conseguir volto aqui e compartilho..
> >
> > Obrigado!

Anderson — *21/06/2019 10:13*

amigo.. do lado SAP tudo ok.. mas não consigo fazer funcionar no ui5 de jeito nenhum...
ja passei pelo erro http 404, 405, tentei algumas alternativas de link e nada...

> Acredito que meu problema seja na montagem correta do link para acessar o connector do scp..
> eu não teria que cadastrar algo no meu neo-app.json? ajuda ai please!!!

> **Richard** — *08/01/2018 14:40*
>
> Legal ver novos posts por aqui!
>
> No aguardo do podcast 😀