

IF140303-Web Application Development

Session-13:

Creating a Live Comments Section Using Websockets

PRU/SPMI/FR-BM-18/0222

Alfa Yohannis



Introduction to Live Comments Section

- In this session, we will implement a live comments section for topics.
- Users will be able to add comments that instantly appear on all other users' screens.
- This will function similarly to a live chat.
- We will utilize WebSockets to achieve this functionality.

What is a WebSocket?

- WebSockets provide a persistent connection between the client and server.
- This allows for real-time, two-way communication.
- Unlike HTTP, WebSockets maintain an open connection, allowing data to be sent and received without repeatedly establishing connections.
- Ideal for applications requiring real-time updates, such as live chats or live notifications.

Flow of the Comments Section

- ****User fills out comment form.****
- ****Clicks submit button.****
- ****WebSocket emits event to the server.****
- ****Server catches the event.****
- ****Server creates a new comment in the database.****
- ****Server emits an event with the updated list of comments.****
- ****All connected clients receive the update and display the new comment.****

Updating the TopicController

- We need to update the 'TopicController' to include a 'show' action.
- The 'show' action retrieves a specific topic by its ID.
- It then renders the 'show.html' template, passing the retrieved topic to the view.
- No need to update the router since we are using 'resources "/"' which already includes a route for 'show'.

Code for the Updated TopicController

```
1  defmodule Discuss.TopicController do
2    use Discuss.Web, :controller
3
4    alias Discuss.Topic
5
6    plug Discuss.Plugs.RequireAuth when action in
7      [:new, :create, :edit, :update, :delete]
```

Code for the Updated TopicController

```
1  plug :check_topic_owner when action in  
2    [:update, :edit, :delete]  
3  
4  def show(conn, %{"id" => topic_id}) do  
5    topic = Repo.get!(Topic, topic_id)  
6    render conn, "show.html", topic: topic  
7  end
```

show.html.eex Template

1

```
<%= topic.title %>
```

- This simple template will display the title of the topic.
- It will serve as the foundation for our live comments section.

Making Topics Clickable in index.html.eex

```
1      <h5>Topics</h5>
2      <ul class = "collection">
3      <%= for topic <- @topics do%>
4      <li class="collection-item">
5      <%= link topic.title, to: topic_path(@conn, :show, topic) %>
6      <%= if @conn.assigns.user.id == topic.user_id do %>
```

Making Topics Clickable in index.html.eex

```
1 <div class="right">
2   <%= link "Edit", to: topic_path(@conn, :edit, topic) %>
3   <%= link "Delete", to: topic_path(@conn, :delete , topic),
      method: :delete %>
4 </div>
5 <% end %>
6 </li>
7 <%end%>
8 </ul>
```

- This makes each topic in the list clickable, linking to its 'show' page.

Add Comments Migration

```
1  defmodule Discuss.Repo.Migrations.AddComments do
2    use Ecto.Migration
3
4    def change do
5      create table(:comments) do
6        add :content, :string
7        add :user_id, references(:users)
8        add :topic_id, references(:topics)
9        timestamps()
10     end
11     ...
```

- Creates a 'comments' table with fields for 'content', 'user_id', and 'topic_id'.
- Establishes relationships between comments , users, and topics.

Creating the Comment Model in comment.ex

```
1  defmodule Discuss.Comment do
2    use Discuss.Web, :model
3
4    schema "comments" do
5      field :content, :string
6      belongs_to :user, Discuss.User
7      belongs_to :topic, Discuss.Topic
8      timestamps()
9    end
```

Creating the Comment Model in comment.ex

```
1  def changeset(struct, params \\ %{}) do
2    struct
3    |> cast(params, [:content])
4    |> validate_required([:content])
5  end
6  end
```

- This model represents the 'comments' table.
- Includes fields for content, user, and topic relationships.
- The 'changeset' function ensures that the content is present when creating or updating a comment.

Updating topic.ex and user.ex Models

```
1 # Add this to both topic.ex and user.ex models
2 has_many :comments, Discuss.Comment
```

- Establishes a one-to-many relationship between topics and comments.
- Similarly, it connects users to their respective comments.

Working with WebSocket: Server and Client Side

- We will work on both server-side and client-side to implement WebSockets.
- WebSockets in Phoenix are handled using channels.
- Channels are an abstraction on top of WebSockets, making it easier to work with real-time features.
- They allow for joining, leaving, and broadcasting messages to groups of users.

How WebSocket Works in Phoenix



- ****Server-side:**** Defines channels that handle different types of messages.
- ****Client-side:**** JavaScript code connects to the server via WebSockets, joins channels, and sends/receives messages.
- This setup ensures real-time communication between all connected clients and the server.

Creating a Comment Channel

- We'll create a new channel to handle comments.
- This will manage events like adding new comments and broadcasting them to all connected clients.
- The channel will be defined in 'comment_channel.ex'.

comment_channel.ex

```
1  defmodule Discuss.CommentChannel do
2    use Discuss.Web, :channel
3    alias Discuss.{Repo, Comment}
4
5    def join("comments:" <> topic_id, _params, socket) do
6      topic_id = String.to_integer(topic_id)
7      comments = Repo.all(
8        from c in Comment,
9        where: c.topic_id == ^topic_id,
10       order_by: [desc: c.inserted_at],
11       limit: 100,
12       preload: [:user]
13     )
```

comment_channel.ex

```
1      {:ok, %{comments: comments}, assign(socket, :topic_id,  
2          topic_id))  
3  
4      def handle_in("new_comment", %{"content" => content}, socket  
5          ) do  
6          topic_id = socket.assigns.topic_id  
7          user_id = socket.assigns.user_id  
          changeset = Comment.changeset(%Comment{topic_id: topic_id,  
              user_id: user_id}, %{"content" => content})
```

comment_channel.ex

```
1 case Repo.insert(changeset) do
2   {:ok, comment} ->
3     broadcast!(socket, "new_comment", %{comment: comment})
4     {:reply, :ok, socket}
5   {:error, _reason} ->
6     {:reply, :error, socket}
7 end
8 end
9 end
```

- Defines a 'join/3' function for users to connect to a specific topic's comments channel.
- The 'handle_in/3' function handles the "new_comment" event, adding the comment to the database and broadcasting it to all clients.

Updating the Socket to Include the Channel

```
1  defmodule Discuss.UserSocket do
2    use Phoenix.Socket
3    channel "comments:*", Discuss.CommentChannel
4    transport :websocket, Phoenix.Transports.WebSocket
5
6    def connect(_params, socket) do
7      {:ok, socket}
8    end
```

Updating the Socket to Include the Channel

```
1  def id(_socket), do: nil  
2  end
```

- Adds the 'comments:*' channel to the socket.
- This ensures that any user connecting to the socket can join the comments channel.

Client-side: Connecting to the WebSocket

```
1 import {Socket} from "phoenix"
2
3 let socket = new Socket("/socket", {params: {userToken: window
4   .userToken}})
5
6 socket.connect()
7
8 let channel = socket.channel("comments:" + topicId, {})
9 channel.join()
   .receive("ok", resp => { console.log("Joined successfully",
   resp) })
```

Client-side: Connecting to the WebSocket

```
1      .receive("error", resp => { console.log("Unable to join",  
2          resp) })  
3  
4      channel.on("new_comment", (payload) => {  
5          let commentContainer = document.querySelector("#comments")  
6          commentContainer.innerHTML += `

${payload.comment.  
            content}</p>`  
        })


```

- Establishes a connection to the WebSocket and joins the comments channel.
- Listens for the "new_comment" event and updates the DOM with the new comment.

Adding a Comment Form in show.html.eex

```
1 <div id="comments">
2   <%= for comment <- @comments do %>
3     <p><%= comment.content %></p>
4   <% end %>
5 </div>
6
7 <form id="comment-form">
8   <input type="text" id="comment-input" placeholder="Add a
   comment..." />
```

Adding a Comment Form in show.html.eex

```
1 <button type="submit">Submit</button>  
2 </form>
```

- Renders existing comments and a form for submitting new ones.
- The form submits comments to the WebSocket channel.

Handling Form Submission in the Client

```
1 let form = document.querySelector("#comment-form")
2 let commentInput = document.querySelector("#comment-input")
3
4 form.addEventListener("submit", (e) => {
5   e.preventDefault()
6
7   let payload = {content: commentInput.value}
8   channel.push("new_comment", payload)
9   .receive("ok", (resp) => { commentInput.value = "" })
```

Handling Form Submission in the Client

```
1      .receive("error", (resp) => { console.log("Failed to send  
2      message", resp) })  
      })
```

- Handles the form submission event.
- Sends the comment content to the WebSocket channel.
- Clears the input field on success or logs an error on failure.

Socket.js: Managing WebSocket Connections

```
1 import {Socket} from "phoenix"
2 let socket = new Socket("/socket", {params: {token: window.
  userToken}})
3
4 socket.connect()
5
6 let channel = socket.channel("comments:1", {})
7 channel.join()
8 .receive("ok", resp => {console.log("Joined successfully",
  resp)}))
```

Socket.js: Managing WebSocket Connections

```
1      .receive("error", resp => {console.log("Unable to join",  
2          resp)})  
3      export default socket
```

- Socket.js sets up the WebSocket connection using Phoenix's JavaScript library.
- The Socket instance is created with the user's token for authentication.
- The `channel.join()` method attempts to join a channel (e.g., "comments:1").
- Success or error messages are logged to the console, helping with debugging.

UserSocket: Handling Channel Connections

```
1  defmodule Discuss.UserSocket do
2    use Phoenix.Socket
3
4    channel "comments:*", Discuss.CommentsChannel
5
6    transport :websocket, Phoenix.Transport.WebSocket
7
8    def connect(_params, socket) do
9      {:ok, socket}
10    end
```

UserSocket: Handling Channel Connections

```
1 def id(_socket), do: nil  
2 end
```

- The UserSocket module defines how the server handles incoming WebSocket connections.
- Channels matching the pattern "comments:*" are routed to Discuss.CommentsChannel.
- The connect/2 function authenticates and establishes the socket connection.
- id/1 returns nil, meaning this socket won't be identifiable for targeted broadcasts.

CommentsChannel: Basic Setup

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3
4    def join(name, _params, socket) do
5      {:ok, %{hey: "there"}, socket}
6    end
```

CommentsChannel: Basic Setup

```
1  def handle_in() do
2      end
3  end
```

- CommentsChannel handles messages and events for the comments section.
- The join/3 function confirms the connection and can send initial data to the client.
- The handle_in/3 function will process incoming events from the client.

Updating app.js

```
1   import "phoenix_html"  
2   import "./socket"
```

- The app.js file imports necessary dependencies for the front-end, including Phoenix HTML helpers and the custom WebSocket logic from socket.js.

WebSocket Join Flow in Phoenix

- Browser initiates connection to the `"comments:1"` channel.
- Server routes the connection through `UserSocket` to `CommentsChannel`.
- `CommentsChannel`'s `join` function is called, returning a response.
- Browser receives the response, triggering success or failure handlers.

Handling Incoming Events

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3
4    def join(name, _params, socket) do
5      {:ok, %{hey: "there"}, socket}
6    end
```

Handling Incoming Events

```
1  def handle_in(name, message, socket) do
2    {:reply, :ok, socket}
3  end
4  end
```

- `handle_in/3` handles incoming events, such as a user submitting a comment.
- It processes the event and sends a reply back to the client, indicating success or failure.

Integrating WebSocket in Views

```
1      <%= @topic.title %>
2      <script>
3      document.addEventListener("DOMContentLoaded", function() {
4          window.createSocket(<%= @topic.id %>)
5      });
6      </script>
```

- This script in `show.html.eex` triggers the WebSocket connection when the page loads.
- The `createSocket` function is called with the topic ID, establishing the channel connection.

Refining socket.js

```
1 import {Socket} from "phoenix"
2 let socket = new Socket("/socket", {params: {token: window.
   userToken}})
3
4 socket.connect()
5
6 const createSocket = (topicId) => {
7   let channel = socket.channel(`comments:${topicId}`, {})
8   channel.join()
9   .receive("ok", resp => {console.log("Joined successfully",
   resp)})
10  .receive("error", resp => {console.log("Unable to join",
   resp)})
11 }
```


Refining socket.js

1

```
window.createSocket = createSocket;
```

- The createSocket function is updated to accept a topicId, dynamically connecting to the correct channel.
- This makes the WebSocket connection context-aware, depending on the topic being viewed.

Updating CommentsChannel with Topic Handling

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3    alias Discuss.Topic
4
5    def join("comments:" <> topic_id, _params, socket) do
6      topic_id = String.to_integer(topic_id)
7      topic = Repo.get(Topic, topic_id)
8
9      {:ok, %{}, socket}
10   end
```

Updating CommentsChannel with Topic Handling

```
1  def handle_in(name, %{"content" => content}, socket) do
2    {:reply, :ok, socket}
3  end
4  end
```

- The `join/3` function now retrieves the topic from the database based on the ID.
- This allows for topic-specific operations within the channel.
- `handle_in/3` processes incoming messages, such as adding a new comment.

Enhancing show.html.eex

```
1    <h5> <%= @topic.title %> </h5>
2
3    <div class="input-field">
4      <textarea class="materialize-textarea"></textarea>
5      <button class="btn">Add Comment</button>
6      <script>
7        document.addEventListener("DOMContentLoaded", function() {
8          window.createSocket(<%= @topic.id %>)
9        });
10     </script>
```

Enhancing `show.html.eex`

- The updated HTML structure in `show.html.eex` includes a form for adding comments.
- The `createSocket` function is triggered when the page loads, setting up the WebSocket connection.

Updating socket.js for Comment Submission

```
1 import {Socket} from "phoenix"
2 let socket = new Socket("/socket", {params: {token: window.
  userToken}})
3
4 socket.connect()
5
6 const createSocket = () => {
7   let channel = socket.channel(`comments:${topicId}`, {})
8   channel.join()
9   .receive("ok", resp => {console.log("Joined successfully",
    resp);})
```

Updating socket.js for Comment Submission

```
1      .receive("error", resp => {console.log("Unable to join",  
2          resp);});  
3  
4      document.querySelector('button').addEventListener('click',  
5          () => {  
6              const content = document.querySelector('textarea').value  
7              ;  
8              channel.push('comment:add', {content: content});  
9          });  
10     }  
  
11     window.createSocket = createSocket;
```

Updating socket.js for Comment Submission

- The `createSocket` function is enhanced to handle user interaction.
- When the button is clicked, the content of the textarea is pushed to the channel.
- This push event triggers a message to be sent to the server, where it will be processed.

Updating CommentsChannel to Handle Comment Submission

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3    alias Discuss.{Topic, Comment}
4
5    def join("comments:" <> topic_id, _params, socket) do
6      topic_id = String.to_integer(topic_id)
7      topic = Repo.get(Topic, topic)
8
9      {:ok, %{}}, assign(socket, :topic, topic)}
10   end
```

Updating CommentsChannel to Handle Comment Submission

```
1  def handle_in(name, %{"content" => content}, socket) do
2    topic = socket.assigns.topic
3    changeset = topic
4    |> build_assoc(:comments)
5    |> Comment.changeset(%{content: content})
6
7    case Repo.insert(changeset) do
8      {:ok, comment} ->
9      {:reply, :ok, socket}
```

Updating CommentsChannel to Handle Comment Submission

```
1      {:error, _reason} ->  
2      {:reply, {:error, %{errors: changeset}}, socket}  
3      end  
4      end  
5      end
```

- The `join/3` function assigns the topic to the socket for future use.
- `handle_in/3` processes the `comment:add` event, creating a new comment in the database.
- The `changeset` validates the comment before it is inserted. If successful, a confirmation is sent back to the client.

Example Flow: Submitting a Comment

- ****Browser****: The client app starts with `topic_id = 1` and joins the channel `"comments:1"`.
- ****Server****: The socket is forwarded to `CommentsChannel`, which sends back the current list of comments.
- ****Browser****: The JavaScript app renders the list of comments, displaying them to the user.

Updating CommentsChannel with Preloading

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3    alias Discuss.{Topic, Comment}
4
5    def join("comments:" <> topic_id, _params, socket) do
6      topic_id = String.to_integer(topic_id)
7      topic = Topic
8      |> Repo.get(topic_id)
9      |> Repo.preload(:comments)
```

Updating CommentsChannel with Preloading

```
1      {:ok, %{comments: topic.comments}, assign(socket, :topic,  
2      topic)}  
3  
4      def handle_in(name, %{"content" => content}, socket) do  
5      topic = socket.assigns.topic  
6      changeset = topic  
7      |> build_assoc(:comments)  
8      |> Comment.changeset(%{content: content})
```

Updating CommentsChannel with Preloading

```
1 case Repo.insert(changeset) do
2   {:ok, comment} ->
3     {:reply, :ok, socket}
4   {:error, _reason} ->
5     {:reply, {:error, %{errors: changeset}}}, socket}
6 end
7 end
8 end
```

- The join/3 function now preloads the comments associated with the topic.
- This allows the server to send all existing comments to the client when the channel is joined.

Updating Comment Model: JSON Serialization

```
1  defmodule Discuss.Comment do
2    use Discuss.Web, :model
3
4    @derive {Poison.Encoder, only: [:content]}
5
6    schema "comments" do
7      field :content, :string
8      belongs_to :user, Discuss.User
9      belongs_to :topic, Discuss.Topic
10     timestamps()
11   end
```


Updating Comment Model: JSON Serialization

```
1 def changeset(struct, params \\ %{}) do
2   struct
3   |> cast(params, [:content])
4   |> validate_required([:content])
5 end
6 end
```

- The Comment model is updated to automatically encode its data to JSON using Poison.
- Only the content field will be included in the JSON output.
- This is useful for sending comment data back to the client in a format that can be easily consumed.

Rendering Comments in socket.js

```
1 import {Socket} from "phoenix"
2 let socket = new Socket("/socket", {params: {token: window.
   userToken}})
3
4 socket.connect()
5
6 const createSocket = () => {
7   let channel = socket.channel(`comments:${topicId}`, {})
8   channel.join()
9   .receive("ok", resp => {
10     renderComments(resp.comments);
11   })
```

Rendering Comments in socket.js

```
1      .receive("error", resp => {console.log("Unable to join",  
2          resp);});  
3  
4      document.querySelector('button').addEventListener('click',  
5          () => {  
6              const content = document.querySelector('textarea').value  
7              ;  
              channel.push('comment:add', {content: content});  
          });  
      };
```

Rendering Comments in socket.js

```
1 function renderComments(comments){
2   const renderedComments = comments.map(comment => {
3     return `
4     <li class="collection-item">
5       ${comment.content}
6     </li>
7     `;
8   });
9
10  document.querySelector('.collection').innerHTML =
11    renderedComments.join('');
```

Rendering Comments in socket.js

1

```
window.createSocket = createSocket;
```

- The `renderComments` function dynamically generates HTML for each comment and inserts it into the DOM.
- This allows the client to display comments immediately after joining the channel.

Updating show.html.eex for Comment Display

```
1 <h5> <%= @topic.title %> </h5>
2
3 <div class="input-field">
4 <textarea class="materialize-textarea"></textarea>
5 <button class="btn">Add Comment</button>
6 <ul class="collection">
7 </ul>
8 <script>
9 document.addEventListener("DOMContentLoaded", function() {
10     window.createSocket(<%= @topic.id %>)
11 });
12 </script>
```

Updating `show.html.eex` for Comment Display

- The HTML template includes a list element to display the comments.
- When the page loads, the `createSocket` function is called, which fetches and renders the comments.

Broadcasting New Comments in CommentsChannel

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3    alias Discuss.{Topic, Comment}
4
5    def join("comments:" <> topic_id, _params, socket) do
6      topic_id = String.to_integer(topic_id)
7      topic = Topic
8      |> Repo.get(topic_id)
9      |> Repo.preload(:comments)
10
11      {:ok, %{comments: topic.comments}, assign(socket, :topic,
12                                                topic)}
13    end
```


Broadcasting New Comments in CommentsChannel

```
1  def handle_in(name, %{"content" => content}, socket) do
2    topic = socket.assigns.topic
3    changeset = topic
4    |> build_assoc(:comments)
5    |> Comment.changeset(%{content: content})
6
7    case Repo.insert(changeset) do
8      {:ok, comment} ->
9        broadcast!(socket, "comments:#{socket.assigns.topic.id}:new
          ", %{comment: comment})
```

Broadcasting New Comments in CommentsChannel

```
1  {:reply, :ok, socket}  
2  {:error, _reason} ->  
3  {:reply, {:error, %{errors: changeset}}}, socket}  
4  end  
5  end  
6  end
```

- The broadcast!/3 function sends the new comment to all clients subscribed to the topic.
- This ensures real-time updates of comments across all clients.

Update socket.js - Adding Event Listener

- We update 'socket.js' to include an event listener for new comments.
- After joining the channel, we listen for "comments:\$topicId:new" events.
- The new comment is rendered on the client side.

```
1 import {Socket} from "phoenix"  
2 let socket = new Socket("/socket", {params: {token: windows.  
   userToken}})  
3  
4 socket.connect()
```

Update socket.js - Adding Event Listener

```
1   const createSocket = () => {  
2     let channel = socket.channel(`comments:${topicId}`, {})  
3     channel.join()  
4     .receive("ok", resp => {  
5       renderComments(resp.comments);  
6     })  
7     .receive("error", resp => {console.log("Unable to join",  
8       resp);  
9     });  
10    channel.on(`comments:${topicId}:new`, renderComment);
```

Update socket.js - Adding Event Listener

```
1      document.querySelector('button').addEventListener('click', ()  
2          => {\n  
3          const content = document.querySelector('textarea').value;  
4          channel.push('comment:add', {content: content});  
5      });  
6  
7      function renderComments(comments){  
8          const renderedComments = comments.map(comment => {\n  
9              return commentTemplate(comment)  
10          });
```

Update socket.js - Adding Event Listener

```
1      document.querySelector('.collection').innerHTML =
        renderedComments.join('');
2    }
3
4    function renderComment(event){
5      const renderedComment = commentTemplate(event.comment)
6
7      document.querySelector('.collection').innerHTML +=
        renderedComment;
8    }
9
10   window.createSocket = createSocket;
```

Authentication with Sockets

- Overview of the authentication flow:
- **Server:** Generate a unique token and add it to the layout.
- **Browser:** Receives the HTML file and boots up the socket, sending the user token.
- **Server:** Verifies the token and assigns the user to the socket.

Update app.html.eex - Adding Unique Token

- We add a script to include the user token if the user is logged in.
- The token is generated using 'Phoenix.Token.sign'.

```
1 <head>
2 <title>Hello Discuss!</title>
3 ...
4 <link rel="stylesheet" href="https://fonts.googleapis.com/
  icon?family=Material+Icons">
```


Update app.html.eex - Adding Unique Token

```
1 <script>
2 <%= if @conn.assigns.user do %>
3   window.userToken = "<%= Phoenix.Token.sign(Discuss.Endpoint,
4     "key", @conn.assigns.user.id) %>"
5 <% end %>
6 </script>
7 ...
</head>
```

Update user_socket.ex

- Using Token for Authentication

- The 'connect' function verifies the token and assigns the user ID to the socket.
- This allows us to authenticate users and maintain user-specific data in the channel.

```
1  defmodule Discuss.UserSocket do
2    use Phoenix.Socket
3
4    channel "comments:", Discuss.CommentsChannel
5
6    transport :websocket, Phoenix.Transport.WebSocket
```

Update user_socket.ex

- Using Token for Authentication

```
1  def connect(%{"token" => token}, socket) do
2    case Phoenix.Token.verify(socket, "key", token) do
3      {:ok, user_id} ->
4        {:ok, assign(socket, :user_id, user_id)}
5      {:error, _error} ->
6        :error
7    end
8  end
9
10 def id(_socket), do: nil
11 end
```

Explanation of build_assoc

- 'build_assoc' is used to create associations between records.
- Limitation: It can only build one relationship at a time and cannot be called twice.
- Example: Assigning a topic to a user by updating 'comments_channel.ex'.

```
1  defmodule Discuss.CommentsChannel do
2    use Discuss.Web, :channel
3    alias Discuss.{Topic, Comment}
4
5    def join("comments:" <> topic_id, _params, socket) do
6      topic_id = String.to_integer(topic_id)
7      topic = Topic
8      |> Repo.get(topic_id)
9      |> Repo.preload(:comments)
```

Explanation of build_assoc

```
1      {:ok, %{comments: topic.comments}, assign(socket, :topic,  
2          topic)}  
3  
4      def handle_in(name, %{"content" => content}, socket) do  
5          topic = socket.assigns.topic  
6          user_id = socket.assigns.user_id  
7  
8          changeset = topic  
9          |> build_assoc(:comments, user_id: user_id)  
10         |> Comment.changeset(%{content: content})
```

Explanation of build_assoc

```
1   case Repo.insert(changeset) do
2     {:ok, comment} ->
3       broadcast!(socket, "comments:#{socket.assigns.topic.id}:new
         ", %{comment: comment})
4     {:reply, :ok, socket}
5     {:error, _reason} ->
6       {:reply, {:error, %{errors: changeset}}}, socket}
7   end
8 end
9 end
```