GO Live Reload with Automatic Page Refresh

https://github.com/dmh2000/go-live-reload

Introduction

If you are coming from the React world of web app development, you probably use 'create-react-app','vite' or some other tool to scaffold out your application. One feature that these provide is live reload of the app when you change something in the source code. It really speeds up development.

In the Go world, you can get live reload of a web app using cosmtrek/air or one of a few other tools : https://techinscribed.com/5-ways-to-live-reloading-go-applications/

THERE IS ONE CATCH WITH GO: In the React environment, when you change code, the server will restart, AND the changes show up immediately on the browser. This is possible because a React app has a bunch of JavaScript running in the app that handle it. Its baked into the React library that is part of the app.

On the other hand, the go tools such as Air will restart your server when source code changes, but they don't force the browser to refresh the current page. So as far as I know with any of the go approaches, you have to manually refresh the page or possibly have it refresh itself with a periodic timer (which is horrible).

There is a solution that seems to work for me. By adding a bit of code, you can have REAL live reload with page refresh on code change.

Approach

1. Setup and use cosmtrek/air to live reload the go web server in the conventional manner. https://github.com/cosmtrek/air
2. Add a tiny piece of JavaScript into your web pages in the Head section. This script creates a simple WebSocket server that listens for a connection from a WebSocket client.

```
<script>
  let active = false;
  sock = new WebSocket("ws://localhost:8080/");
  sock.onopen = function (event) {
    console.log("connected");
    active = true;
  };

  sock.onclose = function (event) {
    console.log("disconnected");
    if (active) {
      setTimeout(function () {
        location.reload();
        active = false;
      }, 2000);
    }
  };
</script>
```

3. In a separate terminal run a small WebSocket client that connects to the server running in the web page. Run this client using Nodemon which is the Node.js tool like Air that restarts a the app when source code change.

```
import { WebSocketServer } from "ws";

const wss = new WebSocketServer({ port: 8080 });

wss.on("open", function connection(ws) {
  console.log("open");
});

wss.on("close", function close() {
  console.log("disconnected");
});

wss.on("error", function error() {
  console.log("error");
});
```

What Does This Do?

Startup

- Open two terminals
- In terminal 1, start the Go web server using Air
    - Air is configured to restart the web server on code change.
    - When the web app first loads, it will enable the WebSocket script which will wait for a connection from the client.
    - Once connected, the WebSocket script does nothing but wait for a disconnect.
- In terminal 2, start the JavaScript client using Nodemon
    - Nodemon is configured to watch the source code of the application
    - It connects to the WebSocket script in the Web page.
- Once the WebSocket endpoints are connected up, there are no messages sent back and forth. There is no extra load on the application. Both ends are passively sitting waiting for a disconnect.

Source Code Change

- Developer changes the code
- In terminal 1, Air will restart the Go web server. It does not cause the web page to reload.
- In terminal 2, Nodemon restarts the WebSocket client. This causes the WebSocket connection to close.
- The WebSocket script in the web page detects a disconnect event.

    - After a slight delay, the script forces a web page reload and the changes show up.

Once the web page reloads and the client restarts, they will reconnect and wait for the next code change.

Note: The setTimeout delay in the scripts makes the web page refresh a little smoother. Without a delay, the browser might momentarily flash a 'site not found' message and then load the web page. It will all work without a delay, but with it you will just see the changes appear. You can tune the delay to match your system response. For me, 1 second wasn't enough, 2 seconds worked.

A Simple Example

Fork this repo to a working directory.

The example code

https://github.com/dmh2000/go-live-reload

Dependencies

Install the following dependencies

- golang
    - go install github.com/cosmtrek/air@latest
- node.js
    - https://nodejs.org
    - download and install
- nodemon
    - https://nodemon.io/
    - npm install -g nodemon

Setup

From root of forked repo:

```
go mod tidy
cd ws
npm install
cd ..
```

Startup

Note : **cmd/main.go** must be run at the top level of the project, not in **cmd**. It needs to access the **views** directory to fetch the HTML template.

Run each of these commands in separate terminals

Start the Golang web app using cosmtrek/air

```

```

```
# live reload of golang web app
$ air
```

Open the browser to see the web page.

Start the websocket app using Nodemon

Nodemon options:

- --**ext** specifies what type of file extension to watch
- --**watch** specifies a directory to watch (can be multiple)

```
# restart the local websocket server so web app will detect disconnect and
reload
$ cd ws
$ nodemon --ext go,mjs,HTML --watch ../cmd --watch ../views  index.mjs

# OR
# the package.json has a start script that does the above
npm start
```

Test

- Make a change to the app source code. Either in cmd/main.go or views/hello.HTML.
- Save the change
- On terminal 1, you should see that Air restarted the Go app
- On terminal 2, you should see that Nodemon restarted the JavaScript app
- On the browser, you should see the changes pop up after a second or two