Trabajo Práctico 5 - Vistas Dinámicas con Templ y SQLc

Objetivo General: Construir una aplicación web básica en Go que renderice datos desde una base de datos utilizando `templ` para la capa de vista y `sqlc` para el acceso a datos, aplicando los conceptos de renderizado en el servidor.

Ejercicio 1: Configuración del Proyecto y SQLc

Objetivo: Preparar el entorno de trabajo, definir un esquema de base de datos simple y generar el código Go para interactuar con ella usando `sqlc`.

1. **Inicializa tu proyecto Go**:
   - Crea un nuevo directorio para el TP.
   - Ejecuta `go mod init <tu-modulo>`.
   - Instala los paquetes necesarios: `go get github.com/a-h/templ` y `go get github.com/mattn/go-sqlite3`.
2. **Define el Esquema de la Base de Datos**:
   - Crea un directorio `db/schema`.
   - Dentro, crea un archivo `schema.sql` con la definición de una tabla de `productos`:
     ```sql
     CREATE TABLE products (
       id INTEGER PRIMARY KEY AUTOINCREMENT,
       name TEXT NOT NULL,
       price REAL NOT NULL
     );
     ```
3. **Define las Consultas SQL**:
   - Crea un directorio `db/queries`.
   - Dentro, crea un archivo `products.sql` con las siguientes consultas:
     ```sql
     -- name: GetProduct :one
     SELECT * FROM products
     WHERE id = ?;

     -- name: ListProducts :many
     SELECT * FROM products
     ORDER BY name;

     -- name: CreateProduct :exec
     INSERT INTO products (name, price)
     VALUES (?, ?);
     ```
4. **Configura `sqlc`**:
   - Crea un archivo `sqlc.yaml` en la raíz del proyecto.
   - Configúralo para que apunte a tu esquema, consultas y genere el código Go en un paquete `db`:
     ```yaml
     version: "2"
     sql:
       - engine: "sqlite"
         queries: "./db/queries"
         schema: "./db/schema"
         gen:
           go:
             package: "db"
             out: "./db/sqlc"
     ```
5. **Genera el código**: Ejecuta `go run github.com/sqlc-dev/sqlc/cmd/sqlc@latest generate` y verifica que se haya creado el directorio `db/sqlc` con el código Go correspondiente.

Ejercicio 2: Creación de Componentes de Vista con Templ

Objetivo: Desarrollar los componentes de la interfaz de usuario utilizando `templ` para crear vistas reutilizables y tipadas.

1. **Crea un directorio `views`** para tus componentes `templ`.
2. **Componente de Layout Básico (`layout.templ`)**:
   - Crea un componente `Layout` que defina la estructura HTML base (`<!DOCTYPE html>`, `<html>`, `<head>`, `<body>`).
   - El layout debe aceptar un `templ.Component` como hijo para renderizar el contenido principal de la página.

- Incluye un link a una hoja de estilos (ej. Pico.css[1]) para una apariencia prolija sin esfuerzo.

3. **Componente para la Lista de Productos (`products.templ`)**:
   - Crea un componente `ProductList` que reciba un slice de `db.Product` (la struct generada por `sqlc`).
   - El componente debe renderizar una tabla (`<table>`) que muestre el nombre y el precio de cada producto.
   - Si la lista está vacía, debe mostrar un mensaje indicándolo.
4. **Componente de Página Principal (`index.templ`)**:
   - Crea un componente `IndexPage` que reciba el título de la página y la lista de productos.
   - Este componente debe usar el `Layout` como base.
   - Dentro del layout, debe renderizar un título `<h1>` y el componente `ProductList`.
5. **Genera el código**: Ejecuta `go run github.com/a-h/templ/cmd/templ@latest generate` y verifica que se hayan creado los archivos `_templ.go`.

Ejercicio 3: Integración en un Servidor Web Go

Objetivo: Unir `sqlc` y `templ` en un servidor HTTP que muestre los datos de la base de datos.

1. **Crea `main.go`** en la raíz del proyecto.
2. **Configura la Base de Datos**:
   - En `main()`, abre una conexión a una base de datos SQLite (`products.db`).
   - Ejecuta el `schema.sql` para crear la tabla si no existe.
   - Crea una instancia de `db.Queries` a partir de la conexión.
3. **Crea el Handler Principal**:
   - Define un handler para la ruta `GET /`.
   - Dentro del handler, usa el método `ListProducts` de `sqlc` para obtener todos los productos.
   - Llama al componente `views.IndexPage`, pasándole los productos obtenidos.
   - Renderiza el componente en el `http.ResponseWriter`.
4. **Inicia el Servidor**: Escucha en el puerto `8080` y registra tu handler.

Ejercicio 4: Creación de Datos con Formularios HTML

Objetivo: Implementar la funcionalidad para agregar nuevos productos a través de un formulario HTML.

1. **Modifica `index.templ`**:
   - Añade un formulario (`<form>`) debajo de la lista de productos.
   - El formulario debe tener campos para el nombre (`name`) y el precio (`price`) del producto.
   - Debe enviar los datos usando el método `POST` a la ruta `/products`.
2. **Crea un Handler para `POST /products`**:
   - Registra una nueva ruta y su handler correspondiente.
   - En el handler, parsea los datos del formulario usando `r.ParseForm()`.
   - Convierte el precio a `float64`.
   - Usa el método `CreateProduct` de `sqlc` para guardar el nuevo producto en la base de datos.
   - Redirige al usuario de vuelta a la página principal (`/`) usando `http.Redirect` para que vea la lista actualizada.

Recursos Adicionales

- Documentación de Templ[2]
- Documentación de SQLc[3]
- Paquete `net/http` de Go[4]
- Driver `go-sqlite3`[5]
- Pico.css (para estilos rápidos)[6]

[1] https://picocss.com/
[2] https://templ.guide/
[3] https://docs.sqlc.dev/
[4] https://pkg.go.dev/net/http
[5] https://github.com/mattn/go-sqlite3
[6] https://picocss.com/

Trabajo de Cursada: Renderizado del Lado del Servidor con Templ

Objetivo: Reemplazar la arquitectura de frontend (HTML/JS/CSS separados) por una aplicación renderizada en el servidor, utilizando `templ` para generar las vistas dinámicamente en Go.

Abandonarás el enfoque de API + Cliente JS para integrar la vista directamente en el servidor Go.

1. **Creación de Componentes `templ`**:
   - Analiza el `index.html` que creaste en el TP4 y divídelo en componentes lógicos:
     - `layout.templ`: La estructura base de la página (HTML, HEAD, BODY), incluyendo un link a una hoja de estilos como Pico.css.
     - `entity_list.templ`: Un componente que recibe un slice de tu entidad (ej. `[]db.Task`) y renderiza la tabla o lista con todos los elementos.
     - `entity_form.templ`: Un componente que renderiza el formulario para crear una nueva entidad.
   - Crea los archivos `.templ` correspondientes en un directorio `views`.
2. **Refactorización del Servidor Go**:
   - Modifica tu `main.go`. Ya no necesitarás los handlers de la API que devuelven JSON.
   - Crea un handler principal para `GET /` que:
     - Obtenga todos los registros de la base de datos usando el método `List...` de `sqlc`.
     - Llame a un componente de página principal (ej. `IndexPage`) que use el layout y renderice el formulario y la lista de entidades (pasándole los datos obtenidos).
     - Renderice el componente completo en el `http.ResponseWriter`.
3. **Manejo del Formulario (Estilo Clásico)**:
   - El `<form>` en tu componente `entity_form.templ` debe hacer un `POST` a una ruta (ej. `POST /<entidades>`).
   - Crea un handler para esta ruta que:
     - Parsee los datos del formulario.
     - Use el método `Create...` de `sqlc` para guardar la nueva entidad en la base de datos.
     - **Redirija** al usuario de vuelta a la página principal (`/`) usando `http.Redirect`. Esto hará que el navegador pida la página de nuevo, y el handler `GET /` mostrará la lista actualizada. Este es el patrón Post/Redirect/Get (PRG).
4. **Generación y Ejecución**:
   - Ejecuta `templ generate` para compilar tus componentes a código Go.
   - Ejecuta tu `main.go` y comprueba que puedes ver la lista de entidades y añadir nuevas a través del formulario, con recargas de página completas.
   - Escribe un `Makefile` o script para automatizar la generación de código y la ejecución del servidor, incluyendo manejo de motor de BD (por ejemplo mediante Docker).