SQLPage Quick Start For Developers

By: Nick Antonaccio (nick@com-pute.com)

See learnsqlpage.com for more details.

Contents:

1. Some Code Examples

SQLPage is an exceptionally productive web development framework which requires only SQL code to create full-stack applications. Here are just a few code examples from this text, running live online:

items.sql	(source)
markdown.sql	(source)
todo.sql	(source)
map_points_of_interest.sql	(source)
animated_webcam_list.sql	(source)
web_cams_db.sql	(source)
shell.sql	(source)
forum.sql	(source)
custom_smiley_card_grid.sql	(source)
upload_file.sql	(source)
iframe.sql	(source)
word_count.sql	(source)
login.sql (joeblow 12341234)	(source)
custom_animated_list.sql	(source)
cash_register.sql	(source)
cash_register_report.sql	(source)
coin_flip.sql	(source)
weather.sql	(source)
image_size.sql	(source)

form_with_table.sql	(source)
url_list_custom.sql	(source)
json_api_table.sql	(source)
sqlalchemy_crud.sql	(source)
contacts_simple.sql	(source)
contacts.sql	(source)

2. Installation and Configuration

Download SQLPage as a single executable program (~8Mb) for Windows, Linux, or Mac:

https://github.com/lovasoa/SQLpage/releases

Unzip or tar -zxvf the downloaded package

Run the SQLPage server program (sqlpage.exe on Windows, sqlpage.bin on Linux and Mac)

Open http://localhost:8080 in your web browser

By default, SQLPage connects to an embedded SQLite database, and serves content on port 8080. Code is saved as .sql files, in the same folder as sqlpage.exe. You can configure these settings by editing the sqlpage.json file in the ./sqlpage/ folder. For example:

```
{
  "database_url": "mssql://sa:thepassword@thedomain.com/thedb",
  "listen_on":  "0.0.0.0:8008",
  "web_root": "./any_folder"
}
```

The database can be sqlite, postgres, mysql, or mssql. All other SQLPage server configuration settings are listed here:

https://github.com/lovasoa/SQLpage/blob/main/configuration.md#configuring-sqlpage

3. UI Components

A core feature of SQLPage is its ability to populate visual UI components with database query results, using only SQL code. To display a visual UI component in a user's browser:

SELECT a SQLPage component

Set properties of the selected component (text content, markdown content, color, layout properties, etc.), using AS clauses
For example, save the code below as 'hello_world.sql', in the same folder as your SQLPage executable:

```
SELECT
    'alert'          AS component,
    'Hello World!'   AS title,
    'black'          AS color;
```

With the SQLPage executable running, open http://localhost:8080/hello_world.sql in your browser. An alert component showing 'Hello World!' in black text is displayed.

See hello_world.sql live online.

3.1 Lists

The SQLPage 'list' UI component displays rows of values queried from a database table. Save the code below as 'list.sql', then (with sqlpage.exe still running) open http://localhost:8080/list.sql in your browser:

```
SELECT
    'list'       AS component,
    'Items:'     AS title;
SELECT
    items        AS title FROM things;
```

See list.sql live online.

In the example above, row values are queried from the 'items' column of a table named 'things'. Note that the title property of the list component above displays the static text value 'Items:', which appears in bold at the top of the component layout. This is called a 'top level' parameter. The other title value, which appears on each line item of the list component, is called a 'row level' parameter. Row-level parameters are populated by the individual values of each row in a database query result.

3.2 Forms

Save the code below as 'form1.sql' and open http://localhost:8080/form1.sql to see it run on your local computer:

```
SELECT 'form'             AS component;   -- a 'form' component is displayed
SELECT 'Item'             AS name;        -- a field named 'Item' is added to the forr

SELECT
```

```
    'alert'                AS component,    -- an 'alert' component is displayed
    'You entered:'         AS title,        -- with this title
    :Item                  AS description   -- and this description
WHERE :Item IS NOT NULL:                     -- only if an Item value has been submitted
```

form1.sql

Note that the single field displayed in the form component above is named 'Item' (by the code: SELECT 'Item' AS name;). When the form is submitted by a user, the POST value is accessed as ':Item'. In this example, that submitted ':Item' value is displayed as the 'description' property of an alert UI component. The alert is only displayed if the :Item value is not null.

3.3 Setting the action property of a form

Submitted form values can be sent to another page for processing, by setting the form component's 'action' property. Save this code as 'form_sender.sql':

```
SELECT
    'form'                 AS component,
    'form_receiver.sql'    AS action;       -- this is the action property
SELECT 'Name'               AS name;
```

And this code as 'form_receiver.sql' (the name of the form action above):

```
SELECT
    'alert'                AS component,
    'You entered:'         AS title,
    :Name                  AS description;
```

form_sender.sql

Note that the submitted value from the form field labeled 'Name', is accessible on the form_receiver.sql page as ':Name'.

As you've seen previously, omitting the action property causes form values to be submitted back to the same page. In that case, the page re-opens and the submitted values can be processed as variables, by prepending any field name with a colon.

3.4 More form examples

Save the code below as 'markdown1.sql' and open http://localhost:8080/markdown1.sql:

```
SELECT
    'form'                   AS component,
    'Enter Markdown Code:'   AS title;
```

```
SELECT
    'Code'                  AS name,
    'textarea'              AS type,
    :Code                   AS value;

SELECT
    'text'                  AS component,
    :Code                   AS contents_md;  -- this property renders markdown content
```

markdown1.sql

Note that the form component above displays a textarea field named 'Code'. When the form is submitted, the posted value from the textarea is accessed as ':Code'. That submitted ':Code' value is displayed as the value property of the textarea (i.e., the submitted code is re-displayed in the textarea field), and also as the markdown content of the text component.

3.5 A full-stack example in 4 lines

In the full-stack example below, the user enters values into a form textarea field, those values are saved to the 'items' column of a database table named 'things', and all the items in that database table are queried and displayed in a UI list component:

```
CREATE TABLE IF NOT EXISTS things (items TEXT NOT NULL);
SELECT 'form' AS component, 'Add item:' AS title; SELECT 'Item' AS name;
INSERT INTO things(items) SELECT :Item WHERE :Item IS NOT NULL;
SELECT 'list' AS component, 'Items:' AS title; SELECT items AS title FROM things;
```

items.sql

Here's a breakdown of how each line works:

A database table is created if it doesn't already exist. The table named 'things' has one column named 'items', which stores text values. Stored text values can not be null.

A form component is shown in the app UI, with the title 'Add item:' and a single text field named 'Item'. When a value is submitted by the user, the page re-opens, with the variable :Item (the name of the form text field) containing the submitted value.

When the app page opens, if an item has been submitted from the form (i.e., the :Item value is not null), then the submitted value is inserted into the 'items' column of the 'things' table.

A list component is shown in the UI, with the top-level title 'Items:'. Rows from the 'items' column of the 'things' table are selected and displayed as the row-level title property of each visual row in the list.

You can format SQLPage code with white space however you prefer. The following code is exactly the same as the 4 lines above:

```
CREATE TABLE IF NOT EXISTS things (
    items        TEXT NOT NULL
);

SELECT
    'form'       AS component,
    'Add item:'  AS title;
SELECT
    'Item'       AS name;

INSERT INTO things(items)
SELECT :Item  WHERE :Item IS NOT NULL;

SELECT
    'list'       AS component,
    'Items:'     AS title;
SELECT
    items        AS title
FROM things;
```

3.6 Adding more form fields and more database columns

The following code follows the same outline as the example above. It simply displays more form fields and list values, which correspond to more columns in a 'myurls' database table. Note that compared to the previous example, the positions of the form and list components are swapped, so the form component now appears visually beneath the table component, but all the other logic and layout structure remain exactly the same (just more components and values):

```
CREATE TABLE IF NOT EXISTS myurls (
    id               INTEGER PRIMARY KEY AUTOINCREMENT,
    url              TEXT NOT NULL,
    description      TEXT NOT NULL
);

INSERT INTO myurls (
    url,
    description
)
SELECT
    :URL,
    :Description
WHERE
    :URL           IS NOT NULL AND
```

```
    :Description   IS NOT NULL;

SELECT
    'list'         AS component,
    'Links:'       AS title;
SELECT
    url            AS title,
    description    AS description,
    url            AS link
FROM myurls;

SELECT
    'form'         AS component,
    'Add link:'    AS title;
SELECT
    'Description'  AS name,
    TRUE           AS required;
SELECT
    'URL'          AS name,
    TRUE           AS required;
```

url_list.sql

Notice that both fields in the form have their 'required' properties set to true (a message is displayed, and the form won't submit until a value is entered for each field).

3.7 UI tables (data grids)

Tables (also called 'data grids') are fundamentally useful in data management applications because they enable users to see and interact with multiple columns and multiple rows of data from a database. The code below displays a UI table component with some static values. Notice that rows of the table below are sortable by clicking column headers, and filterable by typing in the search field.

```
SELECT
    'table'              AS component,
    'Static Data Table'  AS title,
    TRUE                 AS sort,         -- this setting enables column sorting
    TRUE                 AS search;       -- this setting enables filtering
SELECT
    'Alice'              AS 'First Name',
    'Smith'              AS 'Last Name',
    'alice@example.com'  AS 'Email'
SELECT
    'Bob'                AS 'First Name',
    'Jones'              AS 'Last Name',
    'bob@example.com'    AS 'Email';
```

table_example.sql

The displayed values in the example above are not queried from a database table, because there is no FROM clause in the SELECT statements.

3.8 A version of the previous URL links full-stack example, using a table component

The code below simply replaces the UI list component in the previous full-stack 'URL Links' example, with a UI table component:

```
CREATE TABLE IF NOT EXISTS myurls (
    id              INTEGER PRIMARY KEY AUTOINCREMENT,
    url             TEXT NOT NULL,
    description     TEXT NOT NULL
);

INSERT INTO myurls (
    url,
    description
)
SELECT
    :URL,
    :Description
WHERE
    :URL          IS NOT NULL AND
    :Description  IS NOT NULL;

SELECT
    'table'                         AS component,     -- this is the table component
    'Links:'                        AS title,         -- with these properties
    'Link'                          AS markdown,
    TRUE                            AS sort,
    TRUE                            AS search;
SELECT
  description                       AS Description,
  '[' || url || '](' || url || ')'  AS Link
FROM myurls;

SELECT
    'form'          AS component,
    'Add link:'     AS title;
SELECT
    'Description'   AS name,
    TRUE            AS required;
SELECT
    'URL'           AS name,
    TRUE            AS required;
```

url_list_table.sql

Note that the 'Link' column in the table component above is signified to contain markdown content (via the line: 'Link' AS markdown). The displayed markdown content in each row is created by concatenating the url value from the database table with the appropriate markdown code needed to display a link:

```
'[' || url || '](' || url || ')'   AS Link
```

That generates a markdown link in the format [link text](link URL). In this example, the link text and the link URL are the same.

3.9 A more detailed form example

The example below demonstrates a form with more field types, and a variety of useful validation rules:

```
-- Create the demo_users_table if it does not exist
CREATE TABLE IF NOT EXISTS demo_users_table (
    id            INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name    TEXT NOT NULL,
    last_name     TEXT NOT NULL,
    resume        TEXT,
    birth_date    DATE,
    password      TEXT NOT NULL,
    terms         BOOLEAN NOT NULL
);

-- Include the form component with no specified action, for the user to enter new rows
SELECT
    'form'                                          AS component,
    'User'                                          AS title,
    'Create new user'                               AS validate;
SELECT
    'first_name'                                    AS name,
    'John'                                          AS placeholder,
    TRUE                                            AS required;
SELECT
    'last_name'                                     AS name,
    TRUE                                            AS required,
    'We need your last name for legal purposes.'    AS description;
SELECT
    'resume'                                        AS name,
    'textarea'                                      AS type;
SELECT
    'birth_date'                                    AS name,
    'date'                                          AS type,
```

```
    '2010-01-01'                                    AS max,
    '1994-04-16'                                    AS value;
SELECT
    'password'                                      AS name,
    'password'                                      AS type,
    '^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$'        AS pattern,
    TRUE                                            AS required,
    'Minimum eight characters, at least one letter and one number.' AS description;
SELECT
    'I accept the terms and conditions'             AS label,
    'terms'                                         AS name,
    'checkbox'                                      AS type,
    TRUE                                            AS required;

-- Insert submitted form data into database table, if required values are not NULL
INSERT INTO demo_users_table (
    first_name, last_name, resume, birth_date, password, terms
)
SELECT
    :first_name,
    :last_name,
    :resume,
    :birth_date,
    :password,
    CASE WHEN :terms IS NOT NULL THEN 1 ELSE 0 END
WHERE
    :first_name IS NOT NULL AND
    :last_name IS NOT NULL AND
    :password IS NOT NULL;

-- Display a confirmation message
SELECT
    'text' AS component,
    CASE
        WHEN
            :first_name IS NOT NULL AND
            :last_name IS NOT NULL AND
            :password IS NOT NULL THEN
            'New user info has been saved!'
        ELSE
            'No data submitted.'
    END AS contents;

-- Display all rows from demo_users_table in a table
SELECT
    'table'         AS component,
    'Demo Users'    AS title,
    TRUE            AS sort,
    TRUE            AS search;
```

```
SELECT
    id AS 'ID',
    first_name      AS 'First Name',
    last_name       AS 'Last Name',
    resume          AS 'Resume',
    birth_date      AS 'Birth Date',
    -- password     AS 'Password',
    CASE WHEN terms = 1 THEN 'Accepted' ELSE 'Not Accepted' END AS 'Terms Accepted'
FROM demo_users_table;
```

form_with_table.sql

The validation features in the code above can be re-used in many sorts of applications.

3.10 Webcam app

The example below stores webcam info (name, description, url) in a database table named 'webcams'. A form is provided for the user to add new webcam records to the database:

```
CREATE TABLE IF NOT EXISTS webcams (
    id              SERIAL PRIMARY KEY,
    title           TEXT NOT NULL,
    description     TEXT NOT NULL,
    url             TEXT NOT NULL
);

-- Insert data into the webcams table only if the URL is not null
INSERT INTO webcams (title, description, url)
SELECT :title, :description, :url
WHERE :url IS NOT NULL;

-- Form definition for adding a webcam
SELECT
    'form' AS component,
    'Add a webcam:' AS title;
SELECT
    'title' AS name,
    '(a short title)' AS description;
SELECT
    'description' AS name,
    'describe the cam' AS placeholder,
    'textarea' AS type;
SELECT
    'url' AS name,
    'http://' AS value,
    TRUE AS required;

-- Display a datagrid component with the existing webcams
```

```
-- Clicking the link opens the webcam image
SELECT
    'datagrid' AS component,
    'Live Web Cams' AS title;
SELECT
    title AS title,
    description AS description,
    url AS link,
    url AS image_url
FROM webcams;
```

web_cams_db.sql

Click the displayed links to view each live camera image.

4. Logic, Loops, Concatenation, More Components and Other Basic Functionalities

4.1 Conditional evaluations, coin toss app

This example uses CASE to perform branching (if/else) logic. Note the UI card component being used to display the selected image:

```
SELECT
    'card'               AS component;
WITH random_choice       AS (
    SELECT RANDOM() % 2  AS coin_flip
)
SELECT
    CASE
        WHEN coin_flip = 0 THEN 'Heads'
        ELSE 'Tails'
    END AS title,
    CASE
        WHEN coin_flip = 0 THEN 'You win this toss :)'
        ELSE 'You lose this toss :('
    END AS description,
    CASE
        WHEN coin_flip = 0 THEN 'https://re-bol.com/heads.jpg'
        ELSE 'https://re-bol.com/tails.jpg'
    END AS top_image
FROM random_choice;
```

coin_flip.sql

4.2 Loops, 99 Bottles of Beer

SQL CTEs can be used to perform loops within SQLPage queries. Concatenation can be performed using the || operator, or with the Concat() function:

```
WITH RECURSIVE numbers  AS (
    SELECT 99 AS num
    UNION ALL
    SELECT num - 1
    FROM numbers
    WHERE num > 1
)
SELECT
    'text'  AS component,
    num || ' bottles of beer on the wall, ' ||
    num || ' bottles of beer.  Take one down and pass it around, ' ||
    (num - 1) || ' bottles of beer on the wall.' AS contents
FROM        numbers
ORDER BY    num DESC;
```

99_bottles.sql

4.3 CTEs, word count app

SQL CTEs of arbitrary complexity can be incorporated into SQLPage UIs. The following code by Ophir Lojkine counts words submitted in a SQLPage form textarea:

```
SELECT 'form' AS component, 'Enter your text:' AS title;
SELECT 'Text' AS name,
    'textarea' AS type,
    :Text AS value;

WITH RECURSIVE letters(n, l) AS (
  SELECT 0, ''
  UNION ALL
  SELECT n+1, SUBSTRING(:Text, n+1, 1) FROM letters
  WHERE n < LENGTH(:Text)
),
is_word(n, l, is_word) AS (
  SELECT n, l, l BETWEEN 'a' AND 'z' OR l BETWEEN 'A' AND 'Z' FROM letters
),
word_beginnings(is_beginning) AS (
    SELECT is_word AND NOT LAG(is_word) OVER (ORDER BY n) FROM is_word
)
SELECT
    'text' AS component,
    'Word count: ' || SUM(is_beginning::int) AS contents FROM word_beginnings;
```

word_count.sql

4.4 Shell

The shell UI component can be used to frame pages of a SQLPage application, to present menus, headers, footers, and more:

```
SELECT
    'shell'                                 AS component,
    'My page'                               AS title,
    'home'                                  AS icon,
    JSON('{"title": "Main", "submenu": [
        {
            "link":  "/form1.sql",
            "title": "Form",
            "icon":  "forms"
        },
        {
            "link":  "/markdown.sql",
            "title": "Markdown",
            "icon":  "squares"
        },
    ]}')                                    AS menu_item,
    JSON('{"title": "Other", "submenu": [
        {
            "link":  "https://com-pute.com",
            "title": "com-pute.com",
            "icon":  "world"
        },
    ]}')                                    AS menu_item,
    '[Learn](https://learnsqlpage.com)'     AS footer;
```

shell.sql

4.5 More components

There are many more useful UI components built into SQLPage, for example:

```
SELECT
    'chart'             AS component,
    'bar'               AS type,
    'Favorite pets'     AS title;
SELECT
    'Cat'  AS label,
    35     AS value;
SELECT
    'Dog'  AS label,
    45     AS value;
SELECT
```

```
    'Fish' AS label,
    20     AS value;
```

chart1.sql

```
SELECT 'form'       AS component;
SELECT 'latitude'   AS name, '40.6892'   AS placeholder, :latitude   AS value;
SELECT 'longitude'  AS name, '-74.0445'  AS placeholder, :longitude  AS value;

SELECT
    'map' AS component;
SELECT
    'Your pin: ' || :latitude || ',' || :longitude  AS title,
    :latitude    AS latitude,
    :longitude   AS longitude;
```

map2.sql

Save this code as 'page1.sql':

```
SELECT
    'title'            AS component,
    'This is page 1'   AS contents;

SELECT
    'button'           AS component;
SELECT
    '/page2.sql'       AS link,
    'Go to page 2'     AS title;
```

And save this code as 'page2.sql'

```
SELECT
    'title'            AS component,
    'This is page 2'   AS contents;

SELECT
    'button'           AS component;
SELECT
    '/page1.sql'       AS link,
    'Go to page 1'     AS title;
```

page1.sql

Documentation and examples for all the UI components built into SQLPage are available at sql.ophir.dev/documentation.sql.

5. Full CRUD Examples

5.1 ToDo list

In this example, a form is presented for the user to add a new Todo item, then all the saved Todo items are displayed in a UI table component. The user can click the Complete, Edit, and Remove links on each row of the table, to update each row. This example includes the database schema code and everything else needed to create, read, update and delete ToDo records:

```
-- Create table to store ToDo items
CREATE TABLE IF NOT EXISTS todos (
    id            INTEGER PRIMARY KEY,
    task          TEXT NOT NULL,
    completed     BOOLEAN DEFAULT FALSE,
    created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Display a form for adding or editing a ToDo item
SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT
    CASE
        WHEN $edit IS NOT NULL THEN
            (SELECT task FROM todos WHERE id = $edit)
        ELSE ''
    END                          AS value,
    'task'                       AS name,
    'text'                       AS type,
    'Enter or edit your task'    AS placeholder;

-- Insert new task or update existing one
INSERT INTO todos (task)
SELECT :task WHERE :task IS NOT NULL AND $edit IS NULL;

UPDATE todos SET task = :task WHERE id = $edit AND :task IS NOT NULL;

-- Mark task as complete
UPDATE todos SET completed = TRUE WHERE id = $complete;

-- Delete a task
DELETE FROM todos WHERE id = $delete;

-- Display existing ToDo items in a table, along with links to perform database operatic
SELECT 'table'          AS component,
    'ToDo List'         AS title,
    'Completed'         AS markdown,
    'Complete_Action'   AS markdown,
    'Edit_Action'       AS markdown,
    'Remove_Action'     AS markdown,
    TRUE                AS sort,
```

```
    TRUE                AS search;
SELECT
    id                                              AS ID_Number,
    task                                            AS Task,
    CASE WHEN completed THEN 'Yes' ELSE 'No' END    AS Completed,
    '[Complete](todo.sql?complete=' || id || ')'    AS Complete_Action,
    '[Edit](todo.sql?edit=' || id || ')'            AS Edit_Action,
    '[Remove](todo.sql?delete=' || id || ')'        AS Remove_Action
FROM todos:
```

todo.sql

5.2 Using query (URL) parameters

In the ToDo example above, 'query parameter' (GET) values are submitted back to the app from clicked links. Those values, in the form of '?somevariable=somevalue' at the end of URL links, are used to pass row ID numbers to perform Complete, Edit, and Delete actions upon selected rows.

The query parameter values submitted back to the app are constructed using the following concatenated links in the table component:

```
(todo.sql?complete=' || id || ')'
(todo.sql?edit=' || id || ')'
(todo.sql?delete=' || id || ')'
```

Those submitted parameters (represented by ?somevariable=somevalue at the end of URL links), are handled in the receiving code by variable labels prefixed with the $ symbol (dollar sign):

```
$complete
$edit
$delete
```

Throughout the ToDo app logic, whenever one of those query variables has been submitted, their existence is used to conditionally execute SQL statements upon rows of the database table which have matching ID values. For example, if a $delete variable has been submitted, then a DELETE query is run on the todos table, for any row(s) that match the submitted ID:

```
DELETE FROM todos WHERE id = $delete;
```

Similarly, if a $complete variable has been submitted, then an UPDATE query is run on the 'completed' column of the todos table, for any row(s) that match the submitted ID:

```
UPDATE todos SET completed = TRUE WHERE id = $complete;
```

An important clarification to remember is that values submitted via SQLPage forms (using 'POST' method) are accessed using the colon (:) prefix, and values submitted as query parameters in URL links (using 'GET' method) are accessed using the dollar sign ($) prefix.

5.3 Map app

In this app, a database table is used to store latitude and longitude values entered by a user. A UI table component displays all the saved points of interest, with links to remove any selected row and to view any selected point on the map. Those links are passed back to the app URL as query parameters ($delete and $lat $long variables), where conditional evaluations perform specified database queries using the submitted ID value:

```
-- Uncomment this line to delete the database table and start fresh

-- DROP table map_points_of_interest;

CREATE TABLE IF NOT EXISTS map_points_of_interest (
    id INTEGER       PRIMARY KEY AUTOINCREMENT,
    datetime_added   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    latitude         TEXT NOT NULL,
    longitude        TEXT NOT NULL,
    location_name    TEXT NOT NULL
);

DELETE FROM map_points_of_interest WHERE id = $delete;
SELECT 'redirect' AS component, 'map_points_of_interest.sql' AS link
    WHERE $lat IS NOT NULL;

INSERT INTO map_points_of_interest (latitude, longitude, location_name)
SELECT :latitude, :longitude, :location_name
WHERE :latitude IS NOT NULL AND :longitude IS NOT NULL;

SELECT
    'table'               AS component,
    'Points of Interest'  AS title,
    'Remove'              AS markdown,
    'View Point'          AS markdown,
    TRUE                  AS sort,
    TRUE                  AS search;
SELECT
    latitude       AS Latitude,
    longitude      AS Longitude,
```

```
    location_name  AS Name,
    '[Remove](map_points_of_interest.sql?delete=' || id || ')' AS Remove,
    '[Go](map_points_of_interest.sql?lat=' || latitude ||
        '&long=' || longitude || ')' AS 'View Point'
FROM map_points_of_interest;

SELECT
    'map' AS component;
SELECT
    'Your pin: ' || $lat || ',' || $long  AS title,
    $lat    AS latitude,
    $long   AS longitude;

SELECT 'form'          AS component,  'Add a new point of interest:'      AS title;
SELECT 'latitude'      AS name, '40.6892'  AS placeholder, :latitude      AS value;
SELECT 'longitude'     AS name, '-74.0445' AS placeholder, :longitude     AS value;
SELECT 'location_name' AS name, 'Liberty'  AS placeholder, :location_name AS value;
```

map_points_of_interest.sql

5.4 Contacts app, with many typical CRUD features

This example follows code patterns and logic similar to the ToDo app, with form fields to create/update 'name', 'phone', & 'email' values, and a table to view/delete rows:

```
CREATE TABLE IF NOT EXISTS people (
    id            INTEGER PRIMARY KEY,
    name          TEXT,
    phone         TEXT,
    email         TEXT
);

-- Insert new record only if all fields are provided and no edit is in progress
INSERT INTO people (name, phone, email)
SELECT :Name, :Phone, :Email
WHERE :Name IS NOT NULL AND $edit IS NULL;

-- Update the record when editing
UPDATE people
SET name = :Name, phone = :Phone, email = :Email
WHERE id = $edit AND :Name IS NOT NULL;

-- Delete the record
DELETE FROM people WHERE id = $delete;

-- Conditionally show the form for editing or adding a new entry
SELECT 'form' AS component;
-- Populate form fields for both adding and editing
SELECT (SELECT name FROM people WHERE id = $edit)  AS value, 'Name'  AS name;
```

```
SELECT (SELECT phone FROM people WHERE id = $edit) AS value, 'Phone' AS name;
SELECT (SELECT email FROM people WHERE id = $edit) AS value, 'Email' AS name;

-- Add "Add New" button to set the $add parameter
SELECT 'button' as component, 'center' as justify;
SELECT '?add=1' as link, 'Add New' as title;  -- Dynamic link for add new

-- Display the table with actions
SELECT 'table' AS component,
    'Edit' AS markdown, 'Remove' AS markdown, TRUE AS sort, TRUE AS search;
SELECT
    id     AS ID,
    name   AS Name,
    phone  AS Phone,
    email  AS Email,
    '[Edit](?edit=' || id || ')' AS Edit,          -- Dynamic link for edit
    '[🗑](?delete=' || id || ')' AS Remove        -- Dynamic link for delete
FROM people;
```

contacts_simple.sql

Even when displaying thousands of rows, this app performs so quickly that pagination is typically not required. It also performs very fast on low powered budget mobile devices, and in ancient browsers (this app has been tested to work in Retrozilla which runs on Windows95, in the expermental browser on a first generation Kindle, in Dolphin on Android 2.3, in QtWeb 3.8.5, and in many others which don't support modern web UI frameworks). SQLpage's front-end speed and compatibility is impressive.

The version below adds a number of features to the Contacts app. Logic is added to show form fields only when needed (i.e., when adding or updating records). Pagination is added, to show only 100 rows per page, with home/first/next/previous/last links to navigate between pages. An extra form to perform server based filtering has been added, to compliment the table component's ability to search and sort front-end page row results. A confirmation dialogue has also been added to ensure users intend to actually delete selected rows, when they click the trashcan icon:

```
CREATE TABLE IF NOT EXISTS people (
    id INTEGER PRIMARY KEY,
    name TEXT,
    phone TEXT,
    email TEXT
);

-- Define pagination variables
SET $records_per_page = 100;  -- Number of records per page
SET $page = COALESCE($page, 1);  -- Current page number, defaults to 1 if not provided
```

```
SET $offset = ($page - 1) * $records_per_page;  -- Calculate the offset for pagination

-- Insert new record only if all fields are provided and no edit is in progress
INSERT INTO people (name, phone, email)
SELECT :Name, :Phone, :Email
WHERE :Name IS NOT NULL AND $edit IS NULL AND $add IS NOT NULL;

-- Update the record when editing
UPDATE people
SET name = :Name, phone = :Phone, email = :Email
WHERE id = $edit AND :Name IS NOT NULL;

-- Delete the record only if confirmed
DELETE FROM people WHERE id = $delete AND :confirm = 'Yes';

-- Redirect to clear form after insert, update, or deletion confirmation
SELECT 'redirect' AS component, '?' AS link
WHERE
    ($add IS NOT NULL AND :Name IS NOT NULL)  -- Redirect after adding a new record
    OR ($edit IS NOT NULL AND :Name IS NOT NULL)  -- Redirect after editing a record
    OR ($delete IS NOT NULL AND :confirm IS NOT NULL);  -- Redirect after confirming

-- Conditionally show the form if editing or adding a new entry
SELECT 'form' AS component
WHERE $edit IS NOT NULL OR $add IS NOT NULL;

-- Conditionally show the form for confirmation if a deletion is requested and not yet c
SELECT 'form' AS component WHERE $delete IS NOT NULL AND :confirm IS NULL;
SELECT 'hidden' AS type, 'delete' AS name, $delete AS value WHERE $delete IS NOT N
SELECT 'hidden' AS type, 'page' AS name, $page AS value WHERE $delete IS NOT NULL A
SELECT 'hidden' AS type, 'filter' AS name, $filter AS value WHERE $delete IS NOT N
SELECT 'radio' AS type, 'Yes' AS value, 'confirm' AS name, 'Confirm Deletion' AS labe
SELECT 'radio' AS type, 'No' AS value, 'confirm' AS name, 'Do Not Delete' AS label WH

-- Populate form fields for both adding and editing
SELECT
    CASE WHEN $edit IS NOT NULL THEN (SELECT name FROM people WHERE id = $edit) EL
    'Name' AS name
WHERE $edit IS NOT NULL OR $add IS NOT NULL;

SELECT
    CASE WHEN $edit IS NOT NULL THEN (SELECT phone FROM people WHERE id = $edit) EL
    'Phone' AS name
WHERE $edit IS NOT NULL OR $add IS NOT NULL;

SELECT
    CASE WHEN $edit IS NOT NULL THEN (SELECT email FROM people WHERE id = $edit) EL
    'Email' AS name
WHERE $edit IS NOT NULL OR $add IS NOT NULL;
```

```
-- Display the filter form at the top of the page
SELECT 'form' AS component;
SELECT 'text' AS type, 'filter' AS name, 'Filter...' AS placeholder, $filter AS valu
SELECT 'hidden' AS type, 'page' AS name, '1' AS value;  -- Always reset page to 1 on

-- Handle empty filter and create a filter pattern for SQL LIKE
SET $filter = COALESCE($filter, '');  -- Set filter to an empty string if not provide
SET $filter_pattern = '%' || $filter || '%';  -- Create a filter pattern for SQL LIKE

-- Add "Add New" button to set the $add parameter
SELECT 'button' AS component, 'center' AS justify;
SELECT '?add=1&page=' || $page AS link, 'Add New' AS title;

-- Calculate the total number of pages
SET $total_records = (SELECT COUNT(*) FROM people WHERE name LIKE $filter_pattern OR
SET $total_pages = ($total_records + $records_per_page - 1) / $records_per_page;  -- Ca

-- Pagination controls: First, Previous, Next, and Last buttons
SELECT 'text' AS component,
    '[Home](?) | ' ||  -- Home link always appears
    '[First](?page=1&filter=' || $filter || ') | ' ||  -- First link goes to the first
    CASE
        WHEN $page > 1 THEN '[Previous](?page=' || ($page - 1) || '&filter=' || $filte
    END ||
    CASE
        WHEN (SELECT COUNT(*) FROM people WHERE name LIKE $filter_pattern OR phone L
        THEN ' [Next](?page=' || ($page + 1) || '&filter=' || $filter || ') | ' ELSE '
    END ||
    '[Last](?page=' || CAST($total_pages AS INTEGER) || '&filter=' || $filter || ')' AS

-- Display the table with actions and apply pagination
SELECT 'table' AS component,
    'Edit' AS markdown,
    'Remove' AS markdown,
    TRUE AS sort,
    TRUE AS search;

SELECT
    id AS ID,
    name AS Name,
    phone AS Phone,
    email AS Email,
    '[Edit](?edit=' || id || '&page=' || $page || '&filter=' || $filter || ')' AS Edit,
    '[🗑](?delete=' || id || '&page=' || $page || '&filter=' || $filter || '&confirm=n
FROM people
WHERE name LIKE $filter_pattern OR phone LIKE $filter_pattern OR email LIKE $filter_p
LIMIT $records_per_page OFFSET $offset;  -- Apply pagination
```

contacts.sql

This example implements many of the most common features needed in CRUD database interfaces. You can pick and choose between which features to incorporate, in applications of any sort. The online demo of this app uses SQLPage's embedded SQLite database, demonstrating how effective it can be out of the box. Performance can be improved when handling tremendously large data sets (terabytes), and large numbers of users, by connecting to database systems such as Postgres, and by load balancing servers with tools such as nginx.

5.5 Smallest full CRUD datagrid example, a cheatsheet in 8 lines

The following example works the same way as the simple contacts app, but all variables are shortened, extra text is eliminated, and white space formatting is removed. For some minds, this presentation may help simplify SQLPage CRUD code structures:

```
CREATE TABLE IF NOT EXISTS p (i INTEGER PRIMARY KEY, x TEXT, y TEXT);
INSERT INTO p (x, y) SELECT :X, :Y WHERE $e IS NULL AND :X IS NOT NULL;
UPDATE p SET x = :X, y = :Y WHERE i = $e AND :X IS NOT NULL;
DELETE FROM p WHERE i = $d;
SELECT 'button' AS component; SELECT '?' AS link, 'New' AS title;
SELECT 'form' AS component;   -- populate with existing values to edit
  SELECT (SELECT x FROM p WHERE i = $e) AS value, 'X' AS name;
  SELECT (SELECT y FROM p WHERE i = $e) AS value, 'Y' AS name;
SELECT 'table' AS component, 'X' AS markdown, '' AS markdown, TRUE AS sort;
  SELECT i AS I, '['||x||'](?e='||i||')' AS X, y AS Y, '[X](?d='||i||')' AS '' FROM p
```

tight.sql

The following values are represented by the given variables: db-table p, db-fields i x y, form-values :X :Y, UI-columns I X Y '', ?query-parameters $e, $d

Here's an even smaller 8 line version which saves just a single column of values:

```
CREATE TABLE IF NOT EXISTS t (i INTEGER PRIMARY KEY, a TEXT);
INSERT INTO t (a) SELECT :A WHERE $e IS NULL AND :A IS NOT NULL;
UPDATE t SET a = :A WHERE i = $e AND :A IS NOT NULL;
DELETE FROM t WHERE i = $d;
SELECT 'button' AS component; SELECT '?' AS link, 'New' AS title;
SELECT 'form' AS component; SELECT (SELECT a FROM t WHERE i = $e) AS value, 'A' AS
SELECT 'table' AS component, '' AS markdown, 'X' AS markdown, TRUE AS sort;
  SELECT a AS A, i AS I, '[edit](?e='||i||')' AS '', '[x](?d='||i||')' AS X FROM t;
```

cheatsheet.sql

cheatsheet.txt

That's cryptic, but it contains only the necessary code required for all CRUD interactions. It's a useful cheatsheet to keep on hand. Here's a slightly larger version with more readable variables:

cheatsheet_readable.sql
cheatsheet_readable.txt

5.6 Code examples app

This is a simple variation of the contacts app, which displays selected code examples in a code component:

```
CREATE TABLE IF NOT EXISTS code_examples (
    id            INTEGER PRIMARY KEY,
    code          TEXT NOT NULL,
    description   TEXT NOT NULL,
    created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO code_examples (code, description)
SELECT :Code, :Description
WHERE :Code IS NOT NULL AND $edit IS NULL;

UPDATE code_examples
SET code = :Code, description = :Description
WHERE id = $edit AND :Code IS NOT NULL;

DELETE FROM code_examples WHERE id = $delete;

SELECT 'table' AS component,
    'Edit' AS markdown, 'Remove' AS markdown, TRUE AS sort, TRUE AS search;
SELECT
    id           AS ID,
    description  AS Description,
    '[View/Edit](?edit=' || id || ')' AS Edit,
    '[X](?delete=' || id || ')'  AS Remove
FROM code_examples;

SELECT
    'code'          AS component;
SELECT
    description    AS title,
    code           AS contents
FROM code_examples WHERE id = $edit;

SELECT 'button' as component, 'center' as justify;
SELECT '?add=1' as link, 'Add New' as title;
```

```
SELECT 'form' AS component;
SELECT
    (SELECT code FROM code_examples WHERE id = $edit)  AS value,
    'textarea' AS type,
    'Code' AS name;
SELECT
    (SELECT description FROM code_examples WHERE id = $edit) AS value,
    'Description' AS name;
```

http://server.py-thon.com:8008/code_examples.sql

5.7 Forum app

The user interface of the following app example is separated into 2 pages, and the data is saved in joined database tables. This is the home page (forum.sql):

```
-- Table for forum topics
CREATE TABLE IF NOT EXISTS topics (
    id            INTEGER PRIMARY KEY,
    title         TEXT NOT NULL,
    created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table for forum messages
CREATE TABLE IF NOT EXISTS messages (
    id            INTEGER PRIMARY KEY,
    topic_id      INTEGER NOT NULL,
    author        TEXT NOT NULL,
    message       TEXT NOT NULL,
    created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (topic_id) REFERENCES topics(id)
);

-- Form for adding a new topic and initial message
SELECT 'form'     AS component, 'multipart/form-data' AS enctype;
SELECT 'title'    AS name, 'text' AS type, 'Enter topic title' AS placeholder;
SELECT 'author'   AS name, 'text' AS type, 'Your name' AS placeholder;
SELECT 'message'  AS name, 'textarea' AS type, 'Your message' AS placeholder;

-- Insert new topic
INSERT INTO topics (title)
SELECT :title WHERE :title IS NOT NULL;

-- Get the last inserted topic ID
SET $topic_id = (SELECT MAX(id) FROM topics WHERE title = :title);

-- Insert initial message if topic ID is valid
INSERT INTO messages (topic_id, author, message)
SELECT $topic_id, :author, :message
```

```
WHERE :author IS NOT NULL AND :message IS NOT NULL;

-- Display list of topics
SELECT 'table' AS component,
    'Forum Topics'   AS title,
    'View'           AS markdown,
    TRUE             AS sort,
    TRUE             AS search;
SELECT       -- NOTE: the dots join values from the Topics and Messages tables
    t.id             AS ID,
    t.title          AS Topic,
    m.created_at     AS 'Last Message',
    m.last_author    AS 'Last Poster',
    '[View](topic.sql?topic_id=' || t.id || ')'    AS View
FROM topics t
LEFT JOIN (
    SELECT topic_id, MAX(created_at) AS created_at, MAX(author) AS last_author
    FROM messages
    GROUP BY topic_id
) m ON t.id = m.topic_id
ORDER BY m.created_at DESC;
```

This is the topic page (topic.sql):

```
-- Insert reply to the topic
INSERT INTO messages (topic_id, author, message)
SELECT $topic_id, :author, :message
WHERE :author IS NOT NULL AND :message IS NOT NULL;

-- Redirect to refresh the page after posting a reply
SELECT 'redirect'  AS component, 'topic.sql?topic_id=' || $topic_id AS link
WHERE :author IS NOT NULL AND :message IS NOT NULL;

-- Display messages for the selected topic
SELECT 'table' AS component, 'Messages in Topic' AS title, TRUE AS sort, TRUE AS sear
SELECT
    author         AS Author,
    message        AS Message,
    created_at     AS 'Posted At'
FROM messages
WHERE topic_id = $topic_id
ORDER BY created_at ASC;

-- Form for replying to the topic
SELECT 'form'      AS component, 'multipart/form-data' AS enctype;
SELECT 'author'    AS name, 'text' AS type, 'Your name' AS placeholder;
SELECT 'message'   AS name, 'textarea' AS type, 'Your reply' AS placeholder;
```

```
-- Link back to the forum home page
SELECT 'text'      AS component, '[Back to Forum](forum.sql)' AS contents_md;
```

forum.sql

5.8 File management app

This application enables users to upload files from a local device to the SQLPage server, where the files are stored in a database table. A UI table component displays all uploaded file names, the dates they were uploaded, and a download link, which enables users to download any selected file:

```
SELECT
    'form'                  AS component,
    'multipart/form-data'   AS enctype;
SELECT
    'myfile'                AS name,
    'file'                  AS type,
    'File'                  AS label;

WITH uploaded_file AS (
    SELECT
        sqlpage.uploaded_file_name('myfile')  AS fname,
        sqlpage.read_file_as_data_url(sqlpage.uploaded_file_path('myfile'))  AS content
)
INSERT INTO    uploaded_files (fname, content, uploaded)
SELECT         fname, content, CURRENT_TIMESTAMP
FROM           uploaded_file
WHERE          fname IS NOT NULL AND content IS NOT NULL;

SELECT 'text'  AS component, 'File uploaded!'  AS contents
WHERE EXISTS (
    SELECT 1
    FROM uploaded_files
    WHERE fname = sqlpage.uploaded_file_name('myfile')
);

SELECT
    'table'       AS component,
    'Download'    AS markdown,
    TRUE          AS sort,
    TRUE          AS search;
SELECT
    uploaded                                              AS Timestamp,
    fname                                                 AS File_Name,
    '[download](upload_file.sql?filename=' || fname || ')'  AS Download
FROM uploaded_files;
```

```
SELECT 'button'                 AS component;
SELECT
    'Download ' || $filename    AS title,
    content                     AS link FROM uploaded_files
WHERE  $filename IS NOT NULL AND fname = $filename LIMIT 1;
```

upload_file.sql

Note particularly the use of SQLPage functions sqlpage.uploaded_file_name(), sqlpage.read_file_as_data_url(), and sqlpage.uploaded_file_path().

5.9 Cash register and sales report generator

This application is a trivial implementation of a cash register (meant to be a UI demo code example, *not for production use):

```
CREATE TABLE IF NOT EXISTS sales (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    sale_datetime TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    subtotal DECIMAL(10, 2),
    tax DECIMAL(10, 2),
    total DECIMAL(10, 2)
);

CREATE TABLE IF NOT EXISTS line_items (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    sale_id INTEGER,
    description TEXT,
    price DECIMAL(10, 2),
    quantity INTEGER,
    total DECIMAL(10, 2),
    FOREIGN KEY (sale_id) REFERENCES sales(id)
);

-- Remove a line item
DELETE FROM line_items WHERE id = $delete;

-- Redirect to refresh the page after deleting a line item
SELECT 'redirect' AS component, 'cash_register.sql' AS link WHERE $delete IS NOT NUL

-- Form to add a new line item
SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT 'description' AS name, 'text' AS type, 'Item Description' AS placeholder;
SELECT 'price' AS name, 'number' AS type, 'Item Price' AS placeholder, '0.00' AS valu
    0.01 AS step;
SELECT 'quantity' AS name, 'number' AS type, 'Quantity' AS placeholder, '1' AS value;

-- Insert line item
INSERT INTO line_items (description, price, quantity, total)
```

```
SELECT :description, :price, :quantity, :price * :quantity
WHERE :description IS NOT NULL AND :price IS NOT NULL AND :quantity IS NOT NULL;

-- Display line items
SELECT
    'table' AS component,
    'Line Items' AS title,
    'Remove'     AS markdown,
    TRUE AS sort,
    TRUE AS search;
SELECT
    description AS Description,
    printf("%.2f", price) AS 'Price per Item',
    quantity AS Quantity,
    printf("%.2f", total) AS 'Total Cost',
    '[Remove](cash_register.sql?delete=' || id || ')' AS Remove
FROM line_items;

-- Calculate subtotal, tax, and total
SELECT 'text' AS component, 'Subtotal: $' || printf("%.2f", SUM(total))
    AS contents FROM line_items;

SELECT 'text' AS component, 'Tax (10%): $' || printf("%.2f", SUM(total) * 0.1)
    AS contents FROM line_items;

SELECT 'text' AS component, 'Total: $' || printf("%.2f", SUM(total) * 1.1)
    AS contents FROM line_items;

-- Save the transaction
INSERT INTO sales (subtotal, tax, total)
SELECT SUM(total), SUM(total) * 0.1, SUM(total) * 1.1 FROM line_items
WHERE $save IS NOT NULL;

-- Clear line items after saving
DELETE FROM line_items WHERE $save IS NOT NULL;

-- Show a save button
SELECT 'form' AS component;
SELECT 'save' AS name, 'submit' AS type, 'Complete Sale' AS value;
```

cash_register.sql

This sales report app allows users to enter a start date and an end date. When the dates are submitted, the app displays all cash register sales which occurred between the 2 dates, along with a grand total tally of all the saved transactions:

```
-- Form to select date range
SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT 'start_date' AS name, 'date' AS type, 'Start Date' AS label;
```

```
SELECT 'end_date' AS name, 'date' AS type, 'End Date' AS label;

-- Display sales report
SELECT 'table' AS component, 'Sales Report' AS title, TRUE AS sort, TRUE AS search;
SELECT
    sale_datetime AS 'Sale Date',
    printf("%.2f", subtotal) AS 'Subtotal',
    printf("%.2f", tax) AS 'Tax',
    printf("%.2f", total) AS 'Total'
FROM sales
WHERE sale_datetime BETWEEN :start_date AND :end_date
AND subtotal IS NOT NULL;

-- Calculate total sales for the selected date range
WITH total_sales AS (
    SELECT SUM(total) AS total_sales
    FROM sales
    WHERE sale_datetime BETWEEN :start_date AND :end_date
    AND subtotal IS NOT NULL
)
SELECT 'text' AS component, 'Total Sales: $' || printf("%.2f", total_sales)
    AS contents FROM total_sales;
```

cash_register_report.sql

6. Authentication

6.1 Validating hashed passwords

The documentation at https://sql.ophir.dev/documentation.sql?component=authentication#component provides a thorough explanation and examples which demonstrate how to use the SQLPage authentication component.

You can create password hashes with the hash_password function. The simplest possible way to password protect a page is to hard-code a hashed password directly in the application. The app runs the authentication component first, and only continues to display the rest of the page if proper credentials are entered into the authorization popup dialogue:

```
SELECT 'authentication' AS component,
    '$argon2i$v=19$m=8,t=1,p=1$YWFhYWFhYWE$oKBq5E8XFTHO2w' AS password_hash,
    sqlpage.basic_auth_password() AS password;  -- enter 'password'
SELECT
    'alert'              AS component,
    'You are logged in'  AS title,
    'black'              AS color;
```

login_simple.sql (enter 'password' as the password)

This example demonstrates how to create new users and save their hashed passwords to a database table, via an admin page:

```
CREATE TABLE IF NOT EXISTS users(
    id              INTEGER PRIMARY KEY,
    email           TEXT NOT NULL,
    username        TEXT NOT NULL,
    password_hash   TEXT NOT NULL,
    created_at      TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT 'email' AS name, 'email' AS type, 'Enter your email' AS placeholder;
SELECT 'username' AS name, 'text' AS type, 'Enter your username' AS placeholder;
SELECT 'password' AS name, 'password' AS type, 'Enter your password' AS placeholder;

INSERT INTO users (username, email, password_hash)
SELECT :username, :email, sqlpage.hash_password(:password)   -- NOTE THIS LINE
WHERE :username IS NOT NULL AND :password IS NOT NULL AND :email IS NOT NULL;
```

passwords.sql

After user credentials have been entered in the form above, you can provide a form for users to log in to protected pages:

```
SELECT 'form' AS component, 'Log in' AS title, 'authenticate.sql' AS action;
SELECT 'username' AS name, 'text' AS type, 'joebloe' AS placeholder;
SELECT 'password' AS name, 'password' AS type, '12341234' AS placeholder;
```

login.sql

The authentication component can be placed at the top of any page which needs authorization. The form above logs in to this authenticate.sql page:

```
-- Authentication component must be at the top of the page
SELECT 'authentication' AS component,
    (SELECT password_hash FROM users WHERE username = :username) AS password_hash,
    :password AS password;

-- If authentication is successful
SELECT
    'alert'                 AS component,
    'You are logged in'     AS title;
```

6.2 Accessing private data associated with user accounts, using session tokens and cookies

The 4 files below make up a 'private fruits' application, in which each user can only see their own private list of fruits. In the first file, a 'sessions' table schema is created to store session tokens. Save this code as private_fruits_admin.sql. Be sure to open this page to create the new table, before using the rest of the application:

```
CREATE TABLE IF NOT EXISTS private_fruits_users(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS sessions (
    session_token TEXT PRIMARY KEY,
    user_id INTEGER,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES private_fruits_users(id)
);

CREATE TABLE IF NOT EXISTS private_food(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    fruit TEXT NOT NULL,
    user_id INTEGER,
    FOREIGN KEY (user_id) REFERENCES private_fruits_users(id)
);

SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT 'username' AS name, 'text' AS type;
SELECT 'password' AS name, 'password' AS type;

INSERT INTO private_fruits_users (username, password_hash)
SELECT :username, sqlpage.hash_password(:password)
WHERE :username IS NOT NULL AND :password IS NOT NULL;
```

The user page below, named 'private_fruits_login.sql' displays a form, which submits the values entered by the user, to the page 'private_fruits_authenticate.sql':

```
SELECT 'form' AS component,
    'Log in' AS title,
    'private_fruits_authenticate.sql' AS action;
SELECT 'username' AS name, 'text' AS type;
SELECT 'password' AS name, 'password' AS type;
```

After successful authentication, the private_fruits_authenticate.sql code below generates a session token, and saves that random value to the 'sessions' database table. It also saves that same session token to a cookie in the user's browser:

```
SELECT 'authentication' AS component,
    (
        SELECT password_hash
        FROM private_fruits_users
        WHERE username = :username
    ) AS password_hash,
    :password AS password;

-- Generate a session token and cookie after successful authentication
INSERT INTO sessions (session_token, user_id)
VALUES (sqlpage.random_string(32), (
    SELECT id
    FROM private_fruits_users
    WHERE username = :username
))
RETURNING
    'cookie' AS component,
    'session_token' AS name,
    session_token AS value;

SELECT 'redirect' AS component, 'private_fruits.sql' AS link;
```

Finally, on the private_fruits.sql page, the session_token saved in the database is compared with the session_token stored in the user's browser. The key here is that the value stored in any user's browser cookie won't match any other user's randomly generated session token saved in the database. If the user isn't validated properly (by matching a saved session token with a saved browser cookie value), the app redirects to the login page. Otherwise, the rest of the page is displayed:

```
-- Validate session and get user_id
WITH session_data AS (
    SELECT user_id
    FROM sessions
    WHERE session_token = sqlpage.cookie('session_token')
)
-- Redirect to login if session is invalid
SELECT 'redirect' AS component, 'private_fruits_login.sql' AS link
WHERE NOT EXISTS (SELECT 1 FROM session_data);

-- Form for adding a new fruit
SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT 'Fruit' AS name;

-- Insert new fruit for the logged-in user
INSERT INTO private_food(fruit, user_id)
SELECT :Fruit, (
    SELECT user_id
```

```
    FROM sessions
    WHERE session_token = sqlpage.cookie('session_token')
)
WHERE :Fruit IS NOT NULL;

-- Display fruits for the logged-in user
SELECT 'list' AS component, 'My fruit list:' AS title;
SELECT fruit AS title FROM private_food
WHERE user_id = (
    SELECT user_id
    FROM sessions
    WHERE session_token = sqlpage.cookie('session_token')
);
```

The user can now open their personal private fruits page as many times as needed (including upon any form submissions), and the cookie stored in their browser will validate them to access the page, every time. Note that we can place the following part of the code above, on any other page that requires validated access, and the same authorization requirements will be enforced, using the same stored cookie:

```
WITH session_data AS (
    SELECT user_id
    FROM sessions
    WHERE session_token = sqlpage.cookie('session_token')
)
SELECT 'redirect' AS component, 'private_fruits_login.sql' AS link
WHERE NOT EXISTS (SELECT 1 FROM session_data);
```

You can log a user out of the system above by removing the session token from the database table, and removing the session token from the browser cookie:

```
DELETE FROM sessions WHERE session_token = sqlpage.cookie('session_token');
SELECT 'cookie' AS component, 'session_token' AS name, true AS remove;
SELECT 'redirect' AS component, 'private_fruits_login.sql' AS link;
```

7. Accessing 3rd Party REST APIs with SQLPage

7.1 Reading data from web APIs

Below is a simple HTTP request. Response data can be assigned a variable label, or used directly in a SQLPage component. Here the response is displayed in a UI 'code' component:

```
SELECT
    'code'          AS component;
```

```
SELECT
    'API Results'  AS title,
    sqlpage.fetch('https://jsonplaceholder.typicode.com/users')  AS contents;
```

api.sql

Here's another example, which displays a random image from a web API in a SQLPage card component:

```
SELECT
    'card'                          AS component,
    'A random image:'               AS title;
SELECT
    'https://picsum.photos/200'     AS top_image,
    'star'                          AS icon;
```

random_image.sql

This example builds on the random image code above, enabling selection of specifically sized images:

```
SELECT
    'button'                          AS component;
SELECT
    '/image_size.sql?imgsize=200'     AS link,
    '200'                             AS title;
SELECT
    '/image_size.sql?imgsize=400'     AS link,
    '400'                             AS title;
SELECT
    '/image_size.sql?imgsize=800'     AS link,
    '800'                             AS title;

SELECT
    'text'                                                    AS component,
    '![alt text](https://picsum.photos/' || $imgsize || ')'   AS contents_md;
```

image_size.sql

This example by Ophir Lojkine concatenates the base URL of an API at openstreetmap.org, with a search value submitted in query parameter format. The API response to that request contains latitude and longitude values, which are displayed in a SQLPage map component:

```
set url = 'https://nominatim.openstreetmap.org/search?format=json&q=' ||
    sqlpage.url_encode($user_search)
set api_results = sqlpage.fetch($url);
```

```
select 'map' as component;
select $user_search as title,
  CAST($api_results->>0->>'lat' AS FLOAT) as latitude,
  CAST($api_results->>0->>'lon' AS FLOAT) as longitude;
```

Note the CAST operation above, in which the 'lat' and 'lon' values are parsed from the json data returned by the remote API. The ->> operator enables picking from numbered items in list structures, and keys in dictionary structures. JSON data is typically stored as text. The CAST function ensures that these values are treated as numeric types in SQL.

With the code above running at http://server.py-thon.com:8008/searchmap.sql, you could, for example, use the following request with the specified 'eiffel tower' query parameter, to look up the latitude and longitude of the Eiffel Tower and display that location on the map: http://server.py-thon.com:8008/searchmap.sql?user_search=eiffel tower. Try pasting that URL in your browser, and replace 'eiffel tower' with other search values.

7.2 Parsing JSON using database functions

Each RDBMS has different functions for parsing and handling JSON data. SQLite has the built-in functions 'json_each' and 'json_extract', which can be used to parse downloaded json data. This code starts by creating a 'users_parsed' table with 'name', 'username', 'email', and 'company' columns, where select values from the parsed data are stored temporarily. The temporary data in this table is cleared each time the app loads. The json data is then downloaded using the sqlpage.fetch() function, and selected values are stored in the temporary database table. Finally, the database table is queried, and the results are displayed in a UI table component, with the sort and search properties set to TRUE:

```
-- Clear the table at the beginning of the script
DELETE FROM users_parsed;

-- Create table for parsed data
CREATE TABLE IF NOT EXISTS users_parsed (
    id          INTEGER PRIMARY KEY,
    name        TEXT,
    username    TEXT,
    email       TEXT,
    company     TEXT
);

-- Fetch and parse JSON data
WITH raw_data AS (
```

```
    SELECT sqlpage.fetch('https://jsonplaceholder.typicode.com/users/') AS json
),
parsed AS (
    SELECT
        json_extract(value, '$.id')           AS id,
        json_extract(value, '$.name')         AS name,
        json_extract(value, '$.username')     AS username,
        json_extract(value, '$.email')        AS email,
        json_extract(value, '$.company.name') AS company
    FROM raw_data, json_each(raw_data.json)
)
INSERT INTO users_parsed (id, name, username, email, company)
SELECT id, name, username, email, company FROM parsed;

-- Display parsed data
SELECT
    'table'             AS component,
    'Parsed User Data'  AS title,
    TRUE                AS sort,
    TRUE                AS search;
SELECT
    name       AS 'Name',
    username   AS 'Username',
    email      AS 'Email',
    company    AS 'Company'
FROM users_parsed;
```

json_api_table.sql

PostgreSQL has rich JSON support, including the functions jsonb_each_text and jsonb_extract_path_text:

```
-- Assuming you have already fetched the JSON into a variable 'json_data'

WITH raw_data AS (SELECT :json_data::jsonb AS json),
parsed AS (
    SELECT
        (jsonb_each_text(json ->> 'company')).value AS company,
        json->>'id'        AS id,
        json->>'name'      AS name,
        json->>'username'  AS username,
        json->>'email'     AS email
    FROM
        raw_data, jsonb_array_elements(raw_data.json)
)
INSERT INTO users_parsed (id, name, username, email, company)
SELECT id, name, username, email, company FROM parsed
ON CONFLICT (id) DO NOTHING;
```

MySQL supports the JSON functions JSON_UNQUOTE and JSON_EXTRACT:

```
-- Assuming you have already fetched the JSON into a variable 'json_data'

SET @json = :json_data;

INSERT INTO users_parsed (id, name, username, email, company)
SELECT
    JSON_UNQUOTE(JSON_EXTRACT(user.value, '$.id'))           AS id,
    JSON_UNQUOTE(JSON_EXTRACT(user.value, '$.name'))         AS name,
    JSON_UNQUOTE(JSON_EXTRACT(user.value, '$.username'))     AS username,
    JSON_UNQUOTE(JSON_EXTRACT(user.value, '$.email'))        AS email,
    JSON_UNQUOTE(JSON_EXTRACT(user.value, '$.company.name')) AS company
FROM
    JSON_TABLE(@json, '$[*]' COLUMNS (
        value JSON PATH '$'
    )) AS user
ON DUPLICATE KEY UPDATE id=id;
```

MSSQL has the JSON functions JSON_VALUE and OPENJSON:

```
-- Assuming you have already fetched the JSON into a variable 'json_data'

DECLARE @json NVARCHAR(MAX) = :json_data;

INSERT INTO users_parsed (id, name, username, email, company)
SELECT
    JSON_VALUE(user.value, '$.id')           AS id,
    JSON_VALUE(user.value, '$.name')         AS name,
    JSON_VALUE(user.value, '$.username')     AS username,
    JSON_VALUE(user.value, '$.email')        AS email,
    JSON_VALUE(user.value, '$.company.name') AS company
FROM
    OPENJSON(@json) WITH (
        value NVARCHAR(MAX) '$'
    ) AS user
ON CONFLICT (id) DO NOTHING;
```

Keep in mind that ChatGPT and other AI tools can be very helpful in converting specialized SQL code from one database system to another. ChatGPT, Claude, Llama, Deepseek and other AIs do a fantastic job of generating SQL code in general, and you can use them to understand, troubleshoot, and learn anything you need about writing SQLPage apps.

7.3 Composing request bodies for HTTP APIs

This example taken from https://sql.datapage.app/functions.sql?function=fetch#function) demonstrates how to send a more detailed request body to an HTTP endpoint:

```
set request = json_object(
    'method', 'POST'
    'url', 'https://postman-echo.com/post',
    'headers', json_object(
        'Content-Type', 'application/json',
        'Authorization', 'Bearer ' || sqlpage.environment_variable('MY_API_TOKEN')
    ),
    'body', json_object(
        'Hello', 'world',
    ),
);
set api_results = sqlpage.fetch($request);

select 'code' as component;
select
    'API call results' as title,
    'json' as language,
    $api_results as contents;
```

7.4 OpenWeatherMap API example app

This example provides a UI form for the user to enter a zip code. The API at openweathermap.org is then sent a request body containing the zip code, API key, and units of measure. Given that request body, the API returns appropriate information, in json format, about current weather conditions. The wind speed and temperature values picked from the json response data are displayed in a text component. Notice the use of the sqlpage.environment_variable() function, which can be used to keep secrets such as API tokens out of your code:

```
-- Form to enter the zip code
SELECT 'form' AS component, 'multipart/form-data' AS enctype;
SELECT 'zip_code' AS name, 'text' AS type, 'Enter your zip code' AS placeholder;

-- Define the API key
SET $api_key = sqlpage.environment_variable('MY_API_TOKEN');

-- Set a default value for the zip code if not provided
SET $zip_code = COALESCE(:zip_code, '07006');

-- Fetch weather data from OpenWeatherMap API
WITH weather_data AS (
    SELECT sqlpage.fetch(
```

```
        CONCAT(
            'https://api.openweathermap.org/data/2.5/weather?zip=',
            $zip_code,
            '&appid=',
            $api_key,
            '&units=imperial'
        )
    ) AS json_data
)
-- Extract wind speed and temperature
SELECT
    'text' AS component,
    'Wind speed is ' ||
        json_extract(json_data, '$.wind.speed') ||
        ', and temperature is ' ||
        json_extract(json_data, '$.main.temp') AS contents
FROM weather_data;
```

weather.sql

8. Building HTTP API Endpoints in SQLPage

8.1 Returning JSON data structures

An example of a file upload app was presented earlier, which enabled users to upload and save files in the 'content' column of an 'uploaded_files' table in the database, in the following format (the image data is truncated here to save space):

```
 [...]
```

The code below creates an HTTP API endpoint in SQLPage that will deliver a selected image from an SQLite database table. In this example, the file name can be specified in the HTTP request endpoint (the URL where users request a reponse from this app):

```
-- Define the API endpoint
SELECT 'json' AS component,
       json_object(
           'filename', uploaded_files.fname,
           'image_data', uploaded_files.content
       ) AS contents
FROM uploaded_files
WHERE fname = $filename
LIMIT 1;

-- If no file was found, return an error message
SELECT 'json' AS component,
       json_object(
```

```
            'error', 'File not found'
        ) AS contents
WHERE NOT EXISTS (
    SELECT 1
    FROM uploaded_files
    WHERE fname = $filename
);
```

http://server.py-thon.com:8008/image_api.sql?filename=debsmall.jpg

Note that each RDBMS has different built-in functions to handle JSON data:

PostgreSQL:

```
-- Define the API endpoint
SELECT 'json' AS component,
       json_build_object(
           'filename', uploaded_files.fname,
           'image_data', uploaded_files.content
       ) AS contents
FROM uploaded_files
WHERE fname = $filename
LIMIT 1;

-- If no file was found, return an error message
SELECT 'json' AS component,
       json_build_object(
           'error', 'File not found'
       ) AS contents
WHERE NOT EXISTS (
    SELECT 1
    FROM uploaded_files
    WHERE fname = $filename
);
```

MySQL:

```
-- Define the API endpoint
SELECT 'json' AS component,
       JSON_OBJECT(
           'filename', uploaded_files.fname,
           'image_data', uploaded_files.content
       ) AS contents
FROM uploaded_files
WHERE fname = $filename
LIMIT 1;

-- If no file was found, return an error message
SELECT 'json' AS component,
       JSON_OBJECT(
```

```
            'error', 'File not found'
        ) AS contents
WHERE NOT EXISTS (
    SELECT 1
    FROM uploaded_files
    WHERE fname = $filename
);
```

SQL Server:

```
-- Define the API endpoint
SELECT 'json' AS component,
       (SELECT fname AS filename, content AS image_data
        FROM uploaded_files
        WHERE fname = $filename
        FOR JSON PATH, WITHOUT_ARRAY_WRAPPER) AS contents;

-- If no file was found, return an error message
SELECT 'json' AS component,
       (SELECT 'File not found' AS error
        FOR JSON PATH, WITHOUT_ARRAY_WRAPPER) AS contents
WHERE NOT EXISTS (
    SELECT 1
    FROM uploaded_files
    WHERE fname = $filename
);
```

SQLite: Uses json_object() to create the JSON object.

PostgreSQL: Uses json_build_object() to create a JSON object.

MySQL: Uses JSON_OBJECT() to create the JSON object.

SQL Server: Uses FOR JSON PATH, WITHOUT_ARRAY_WRAPPER to create the JSON object.

In each case, the query checks if the file exists and returns the file's content if found, or an error message if not.

A comprehensive document about using JSON in SQLPage is at https://sql.datapage.app/blog.sql?post=JSON%20in%20SQL%3A%20A%20Comprehensive%20Guide

9. Using sqlpage.exec() to Execute Local Code

The sqlpage.exe() function can be used to run compiled code and/or any code run by an interpreter of any programming language, on the server computer. To use this function, set "allow_exec" : true in the sqlpage/sqlpage.json configuration settings file.

9.1 Several Python examples

The following example uses sqlpage.exec() to run a Python script which counts the number of words in a submitted ':text_input' string:

```
-- Form to input text
SELECT 'form' AS component;
SELECT 'text_input' AS name, 'textarea' AS type, :text_input AS value;

-- Execute the Python script to count words
SELECT 'text' AS component,
    'Word Count: ' || sqlpage.exec(
      'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Python311\\python.exe',
      'word_count.py',
      :text_input
    ) AS contents;
```

Here is the Python word_count.py script which is called by the SQLPage code above. The word_count.py file must be placed in the SQLPage working directory (the same folder where sqlpage.exe or is located).:

```
import sys

text = sys.argv[1]
word_count = len(text.split())
print(word_count)
```

The SQLPage app below runs a Python script, which uses the Python 'requests' library to download a joke from https://official-joke-api.appspot.com/random_joke:

```
SELECT 'text' AS component,
    'Here''s a joke: '  AS title,
    sqlpage.exec(
        'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Python311\\python.exe',
        'get_joke.py'
    )                   AS contents;
```

Here's the Python code called in the example above. Save it as get_joke.py in your SQLPage working directory:

```
import requests

response = requests.get('https://official-joke-api.appspot.com/random_joke')
joke_data = response.json()
print(f"{joke_data['setup']} - {joke_data['punchline']}")
```

This SQLPage script fetches and displays a markdown table generated by some Python pandas code:

```
SELECT 'text' AS component,
    sqlpage.exec(
        'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Python311\\python.exe',
        'generate_table.py'
    ) AS contents_md;
```

Here's the Python code. Save it as generate_table.py in your SQLPage working directory:

```
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "City": ["New York", "Los Angeles", "Chicago"]
}
df = pd.DataFrame(data)
markdown_table = df.to_markdown(index=False)
print(markdown_table)
```

The examples in this section are all Python, but sqlpage.exec() can run any code compiled or interpreted by any other programming language (this includes any arbitrary executable or script evaluated by a language interpreter on your server), so incorporating all sorts of existing software tools and collaborating with developers of any backgroung is straightforward. Be sure to understand the security implications before using sqlpage.exec() in any SQLPage application.

10. Integrating with SQLAlchemy

This example doesn't use sqlpage.exec(), but accomplishes a useful task for projects which integrate Python. When creating a database table with the Python SQLAlchemy library like this:

```
from sqlalchemy import create_engine, Column, Integer, String, Date, Enum, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
connect_string="mssql+pyodbc://sa:pass@url.com/db?driver=ODBC+Driver+17+for+SQL+Server"
engine = create_engine(connect_string, echo=False)
Base = declarative_base()
Session = sessionmaker(bind=engine)

class Demog(Base):
```

```
    __tablename__ = 'demog'
    personID = Column(Integer, primary_key=True, autoincrement=True)
    regID = Column(Integer, ForeignKey('users.regID'), nullable=False)
    display = Column(Enum('yes', 'no'), default='no')
    LastName = Column(String(30))
    FirstName = Column(String(30))
    MiddleName = Column(String(25))
    Gender = Column(Enum('Male', 'Female', 'Other'))
    DOB = Column(Date)
    SSN = Column(String(9), default='', nullable=True)
    notes = Column(String(255))

    users = relationship("Users", back_populates="demogs")
```

The following SQLPage example can be used to display a UI table and a form to interact with that database table:

```
INSERT INTO demog (
    regID,
    LastName,
    FirstName,
    MiddleName,
    Gender,
    DOB,
    SSN,
    notes,
    display
)
SELECT
    :regID,
    :LastName,
    :FirstName,
    :MiddleName,
    :Gender,
    :DOB,
    :SSN,
    :notes,
    :display
WHERE
    :regID          IS NOT NULL AND
    :DOB            IS NOT NULL AND
    :LastName       IS NOT NULL AND
    :FirstName      IS NOT NULL;

SELECT
    'table'         AS component,
    'People:'       AS title,
    TRUE            AS sort,
    TRUE            AS search;
```

```
SELECT
    regID           AS regID,
    LastName        AS LastName,
    FirstName       AS FirstName,
    MiddleName      AS MiddleName,
    Gender          AS Gender,
    DOB             AS Date_Of_Birth,
    SSN             AS SSN,
    notes           AS notes,
    display         AS display
FROM demog;

SELECT
    'form'          AS component,
    'Add link:'     AS title;
SELECT
    'regID'         AS name,
    'number'        AS type,
    TRUE            AS required;
SELECT
    'LastName'      AS name,
    TRUE            AS required;
SELECT
    'FirstName'     AS name,
    TRUE            AS required;
SELECT
    'MiddleName'    AS name;
SELECT
    'Gender'           AS name;
SELECT
    'SSN'           AS name;
SELECT
    'notes'         AS name;
SELECT
    'DOB'           AS name,
    'date'          AS type,
    '2010-01-01'    AS max,
    '1990-01-15'    AS value;
SELECT
    'display'       AS name,
    'select'        AS type,
    -- TRUE         AS searchable,
    '[{"label": "no", "value": "no"}, {"label": "yes", "value": "yes"}]' AS options,
    'no'            AS value;
```

Note that for this SQLAlchemy connection string:

```
connect_string="mssql+pyodbc://sa:pass@url.com/db?driver=ODBC+Driver+17+for+SQL+Server"
```

The SQLPage connection string in ./sqlpage/sqlpage.json would be:

```
{"database_url": "mssql://sa:pass@url.com/db"}
```

Here's a full CRUD version of the code above, with a schema definition at the top of the code, so it can also be used anywhere SQLAlchemy hasn't been used to set up the database. Examine the structure of this code and notice that it's derived almost exactly from the Contacts CRUD example provided earlier in the tutorial:

```
-- SQL Schema Definition, if SQLAlchemy hasn't been used to set up the database

CREATE TABLE IF NOT EXISTS demog (
    personID INTEGER PRIMARY KEY AUTOINCREMENT,
    regID INTEGER NOT NULL,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    MiddleName VARCHAR(25),
    Gender VARCHAR(10) CHECK (Gender IN ('Male', 'Female', 'Other')),
    DOB DATE,
    SSN VARCHAR(9) DEFAULT '' NULL,
    notes VARCHAR(255),
    display VARCHAR(3) CHECK (display IN ('yes', 'no')) DEFAULT 'no'
);

-- Create an index on regID if needed
CREATE INDEX IF NOT EXISTS regID_idx ON demog (regID);

-- Insert new record only if no edit is in progress
INSERT INTO demog (
    regID,
    LastName,
    FirstName,
    MiddleName,
    Gender,
    DOB,
    SSN,
    notes,
    display
)
SELECT
    :regID,
    :LastName,
    :FirstName,
    :MiddleName,
    :Gender,
    :DOB,
    :SSN,
    :notes,
```

```
    :display
WHERE
    :regID       IS NOT NULL AND
    :DOB         IS NOT NULL AND
    :LastName    IS NOT NULL AND
    :FirstName   IS NOT NULL AND
    $edit        IS NULL;  -- Only insert if no edit is in progress

-- Update the existing record only if editing
UPDATE
    demog
SET
    regID =       :regID,
    LastName =    :LastName,
    FirstName =   :FirstName,
    MiddleName =  :MiddleName,
    Gender =      :Gender,
    DOB =         :DOB,
    SSN =         :SSN,
    notes =       :notes,
    display =     :display
WHERE
    personID = $edit   AND
    :regID            IS NOT NULL AND
    :DOB              IS NOT NULL AND
    :LastName         IS NOT NULL AND
    :FirstName        IS NOT NULL;

-- Delete the record
DELETE FROM demog WHERE personID = $delete;

-- Form for adding or editing
SELECT
    'form'            AS component,
    'Add a patient:'  AS title;
SELECT
    (SELECT regID FROM demog WHERE personID = $edit)  AS value,
    'regID'         AS name,
    'number'        AS type;
SELECT
    (SELECT LastName FROM demog WHERE personID = $edit)  AS value,
    'LastName'      AS name;
SELECT
    (SELECT FirstName FROM demog WHERE personID = $edit)  AS value,
    'FirstName'     AS name;
SELECT
    (SELECT MiddleName FROM demog WHERE personID = $edit)  AS value,
    'MiddleName'    AS name;
SELECT
```

```
    (SELECT Gender FROM demog WHERE personID = $edit)  AS value,
    'Gender'          AS name;
SELECT
    (SELECT SSN FROM demog WHERE personID = $edit)  AS value,
    'SSN'             AS name;
SELECT
    (SELECT notes FROM demog WHERE personID = $edit)  AS value,
    'notes'           AS name;
SELECT
    (SELECT DOB FROM demog WHERE personID = $edit)  AS value,
    'DOB'             AS name,
    'date'            AS type,
    '2010-01-01'      AS max;
SELECT
    (SELECT display FROM demog WHERE personID = $edit)  AS value,
    'display'         AS name,
    'select'          AS type,
    '[{"label": "no", "value": "no"}, {"label": "yes", "value": "yes"}]' AS options,
    'no'              AS value;

-- Button for adding new entry
SELECT 'button' as component, 'center' as justify;
SELECT '?add=1' as link, 'Add New' as title;

-- Table to display demog records
SELECT
    'table'           AS component,
    'Demog:'          AS title,
    'Edit'            AS markdown,
    'Remove'          AS markdown,
    TRUE              AS sort,
    TRUE              AS search;
SELECT
    personID          AS personID,
    regID             AS regID,
    LastName          AS LastName,
    FirstName         AS FirstName,
    MiddleName        AS MiddleName,
    Gender            AS Gender,
    DOB               AS Date_Of_Birth,
    SSN               AS SSN,
    notes             AS notes,
    display           AS display,
    '[Edit](?edit=' || personID || ')' AS Edit,
    '[X](?delete=' || personID || ')' AS Remove
FROM demog;
```

sqlalchemy_crud.sql

11. Building Custom UI Components

11.1 A simple example

The documentation at https://sql.datapage.app/custom_components.sql explains how new SQLPage UI components can be created. As an example, save the following code as 'smiley_list.handlebars' in the 'sqlpage/templates/' subfolder of your working directory (that should be a subfolder of the directory where your sqlpage.exe binary is located):

```
<h1>{{big_text}}</h1>

<ul>
{{#each_row}}
    <li>😊 {{id}} - {{fruit}}</li>
{{/each_row}}
</ul>
```

Notice that the template above has 3 properties included within double curly brackets {{}}:

big_text

id

fruit

The template also includes a smiley emoji, as well as a dash between the 'id' and 'fruit' properties. That 'smiley_list' component can now be used like any other built-in component that comes with SQLPage. We'll use it to display a list of fruits from the 'food' database table created in an earlier example. Save this code as 'custom_smiley_list.sql' in the same folder where all your other normal SQLPage code application files are located (not in the same folder as the .handlebars file):

```
SELECT 'smiley_list' AS component, 'My smiley fruit list' AS big_text;
SELECT id AS id, fruit AS fruit FROM food;
```

custom_smiley_list.sql

For reference, the code above queries a database table which can be populated with this code:

```
CREATE TABLE food (
    id      INTEGER PRIMARY KEY,
    fruit   TEXT NOT NULL
);
SELECT 'form' AS component, 'Add a fruit' AS title;
```

```
  SELECT 'Fruit' as name,
  TRUE as required;

INSERT INTO food(fruit)
  SELECT :Fruit
  WHERE :Fruit IS NOT NULL;

SELECT 'list' AS component, 'Current fruits' AS title;
  SELECT fruit AS title,
  CONCAT(fruit, ' is a favorite fruit') as description FROM food;
```

fruit1.sql

Try adding a fruit above and then re-running custom_smiley_list.sql.

11.2 Another simple example

Save this custom component code as '/sqlpage/templates/my_urls.handlebars':

```
<h1>{{title}}</h1>

<ul>
{{#each_row}}
    <li>{{description}}: <a href="{{url}}" target=_blank>{{url}}</a></li>
{{/each_row}}
</ul>
```

You can now use that component in any SQLPage application on your server. Here is an updated version of the URL list application from earlier in the tutorial, now using the component above to display the URL list:

```
CREATE TABLE IF NOT EXISTS myurls (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  url TEXT NOT NULL,
  description TEXT NOT NULL
);

INSERT INTO myurls (url, description)
SELECT :URL, :Description
WHERE :URL IS NOT NULL AND :Description IS NOT NULL;

SELECT 'my_urls' AS component, 'Links:' AS title;
SELECT
  url AS url,
  description AS description
FROM myurls;

SELECT 'form' AS component, 'Add link:' AS title;
```

```
SELECT 'Description' AS name, TRUE as required;
SELECT 'URL' AS name, TRUE as required;
```

url_list_custom.sql

11.3 Using Bootstrap and Tabler classes in custom components

SQLPage uses built-in Bootstrap and Tabler styling (Tabler is built on top of Bootstrap), including more than 5000 graphic icons. You can also include custom CSS and Javascript code to adjust spacing, background colors, animations, or any other visual property.

Save the following code as 'smiley_card_grid.handlebars' in the 'sqlpage/templates/' folder:

```
<h1>{{title}}</h1>

<div class="row">
{{#each_row}}
    <div class="col-md-4">
        <div class="card">
            <div class="card-body">
                <div class="card-icon">{{{icon_img "mood-smile"}}}</div>
                <h5 class="card-title">{{author}}</h5>
                <p class="card-text">{{message}}</p>
            </div>
        </div>
    </div>
{{/each_row}}
</div>
```

Save this code as 'custom_smiley_card_grid.sql':

```
SELECT 'smiley_card_grid' AS component, 'Smiley Message Cards' AS title;
SELECT message AS message, CONCAT('Author: ', author) AS author FROM messages;
```

custom_smiley_message_grid.sql

The example above queries the 'messages' database table, which was created for the earlier public forum application example. This example displays all the forum messages using the custom card layout defined above.

11.4 Several custom animated components

The following template animates a list of items, so that each item fades in as it appears. Note that this is all standard CSS code, with placeholders for database values,

enclosed in handlebars notation (double curly brackets: {{}}). Save this code as 'sqlpage/templates/animated_list.handlebars':

```
<h1>{{title}}</h1>

<ul class="animated-list">
{{#each_row}}
    <li style="animation-delay: {{@row_index}}s;">{{first_property}}: {{second_property}
{{/each_row}}
</ul>

<style nonce="{{@csp_nonce}}">
    .animated-list {
        list-style-type: none;
        padding: 0;
    }

    .animated-list li {
        opacity: 0;
        transform: translateY(20px);
        animation: fadeInUp 0.5s ease-out forwards;
    }

    @keyframes fadeInUp {
        to {
            opacity: 1;
            transform: translateY(0);
        }
    }
</style>
```

The SQLPage code below uses the component above to display values from the 'webcams' table created in an earlier application example:

```
SELECT 'animated_list' AS component, 'Animated List' AS title;
SELECT title AS first_property, description AS second_property FROM webcams;
```

custom_animated_list.sql

Here's the webcams table definition, for reference:

```
CREATE TABLE webcams (
    id              SERIAL PRIMARY KEY,
    title           TEXT NOT NULL,
    description     TEXT NOT NULL,
    url             TEXT NOT NULL
);
```

The code below adjusts the animated fade-in list component above to include images from the url field in the webcams database table. Save this template as 'sqlpage/templates/animated_webcam_grid.handlebars':

```
<h1>{{title}}</h1>

<div class="animated-grid">
{{#each_row}}
    <div class="grid-item" style="animation-delay: {{@row_index}}s;">
        <img src="{{url}}" alt="{{title}}" class="webcam-image">
        <h2>{{title}}</h2>
        <p>{{description}}</p>
    </div>
{{/each_row}}
</div>

<style nonce="{{@csp_nonce}}">
    .animated-grid {
        display: flex;
        flex-wrap: wrap;
        gap: 20px;
        justify-content: space-around;
        padding: 0;
    }

    .grid-item {
        width: 200px;
        opacity: 0;
        transform: translateY(20px);
        animation: fadeInUp 0.5s ease-out forwards;
        text-align: center;
    }

    .webcam-image {
        width: 100%;
        height: auto;
        border-radius: 8px;
        margin-bottom: 10px;
    }

    @keyframes fadeInUp {
        to {
            opacity: 1;
            transform: translateY(0);
        }
    }
</style>
```

Using this template in SQLPage apps requires very little code:

```
SELECT 'animated_image_list' AS component, 'Webcam Gallery' AS title;
SELECT title, description, url FROM webcams;
```

custom_animated_image_list2.sql

As you can see, custom components enable complex UI designs and functionality to be encapsulated and then incorporated very simply in SQLPage UI code. This enables easy separation of concerns, and neatly organized collaborative effort between front-end and back-end developers. See http://learnsqlpage.com for a more detailed explanation of these custom components.

12. Incorporating Iframes

iFrames enable web applications running at a remote URL to be embedded in SQLPage apps. iFrames can be displayed in SQLPage card components. For example, this plain HTML code:

```
<h1>Hello World</h1><br>
<a href="hello_world.html" target="_parent">hello world</a><br>
<a href="hello_world_iframe.sql" target="_parent">hello world iframe</a>
```

hello_world.html

Can be displayed by this SQLPage code:

```
SELECT
    'card'                                          AS component,
    'The Hello World HTML page running in an iFrame:' AS title,
    1                                               AS columns;
SELECT
    'The section below could be ANY web app'        AS title,
    'http://server.py-thon.com:8008/hello_world.html' AS embed,
    'iframe'                                        AS embed_mode,
    '350'                                           AS height;
```

http://server.py-thon.com:8008/hello_world_iframe.sql

Applications delivered in iframes can be of any complexity, and can be created using virtually any web development technology:

```
SELECT
    'card'                                          AS component,
    'Multiple remote applications running in iFrames' AS title,
    2                                               AS columns;
```

```
SELECT
    'Message Board (Bootstrap, Plain JS, Bottle)'      AS title,
    'https://server.py-thon.com:8450'                  AS embed,
    'iframe'                                           AS embed_mode,
    '350'                                              AS height;
SELECT
    'Embedded Image Uploader (Brython, Flask)'         AS title,
    'http://216.137.179.125:8023'                      AS embed,
    -- https://server.py-thon.com:8443/?sort_by=name&order=asc
    'iframe'                                           AS embed_mode,
    '350'                                              AS height
SELECT
    '3D Message (NiceGUI)'                             AS title,
    'http://216.137.179.125:9090'                      AS embed,
    'iframe'                                           AS embed_mode,
    '350'                                              AS height
SELECT
    'Editable Cards (Streamlit, Dataset, SQLAlchemy)'  AS title,
    'http://216.137.179.125:8501'                      AS embed,
    'iframe'                                           AS embed_mode,
    '350'                                              AS height

SELECT
    'card'                                             AS component,
    1                                                  AS columns;
SELECT
    'Contacts & Auth: joe@guitarz.org 12341234'        AS title,
    'http://server.py-thon.com:5001'                   AS embed,
    'iframe'                                           AS embed_mode,
    '500'                                              AS height;
```

iframe.sql

12.1 Integrating apps which call SQLPage API endpoints, in SQLPage iframes

Keep in mind that apps written in other programming languages can call APIs written in SQLPage, and vice-versa. This makes iframes a great way to enable easy collaboration with developers who choose to use web APIs in any other programming language ecosystem. For example, the SQLPage API below returns JSON data containing the author and title column values of all messages, from the Forum example app earlier in this tutorial:

```
SELECT 'json' AS component,
    JSON_OBJECT(
        'messages', (
            SELECT JSON_GROUP_ARRAY(
                JSON_OBJECT(
```

```
                'author', author,
                'message', message
            )
        ) FROM messages
    )
) AS contents;
```

http://server.py-thon.com:8008/message_api.sql

The code below uses the JS Fetch API to request data from the API above, and Bootstrap to display those values in a table layout:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
</head>
<body>

<div class="container" style="margin-top: 0;">
    <table class="table table-striped">
        <thead>
            <tr>
                <th scope="col">Author</th>
                <th scope="col">Message</th>
            </tr>
        </thead>
        <tbody id="message-table-body">
            <!-- Rows will be inserted here dynamically -->
        </tbody>
    </table>
</div>

<script>
// Fetch data from the API
fetch('http://server.py-thon.com:8008/message_api.sql')
    .then(response => response.json())
    .then(data => {
        const messages = data.messages;
        const tableBody = document.getElementById('message-table-body');

        messages.forEach(msg => {
            const row = document.createElement('tr');
            row.innerHTML = `<td>${msg.author}</td><td>${msg.message}</td>`;
            tableBody.appendChild(row);
        });
    })
```

```
    .catch(error => console.error('Error fetching the data:', error));
</script>

</body>
</html>
```

http://server.py-thon.com:8008/bootstrap_message_table.html

Finally, this SQLPage code embeds the Bootstrap table layout above in a SQLPage card component:

```
SELECT
    'card'                                              AS component,
    'Bootstrap table running in an iFrame:'             AS title,
    1                                                   AS columns;
SELECT
    'Data from http://server.py-thon.com:8008/message_api.sql'      AS title,
    'http://server.py-thon.com:8008/bootstrap_message_table.html'   AS embed,
    'iframe'                                                        AS embed_mode,
    '350'                                                           AS height;
```

http://server.py-thon.com:8008/bootstrap_message_table_iframe.sql

Those 3 pieces form a fully self-contained app, served entirely by SQLPage, with data returned solely by a SQLPage API. No other programming languages (Python, PHP, Java, etc.), ORMs (SQLAlchemy, pyDAL, Pony, etc.), or any other API server framework (Flask, FastAPI, etc.) are required on the back end. In this way, SQLPage can take the place of all those other complex architecture pieces, and integrate beautifully with any front-end tooling of your choice.

SQLPage applications can also be embedded in web apps created using any other programming language tools. Here's the contacts app from earlier in the tutorial, embedded in a Python Anvil app:

https://sqlpage.anvil.app

As you can see, iframes provide a simple and fast way to integrate complete full-stack apps directly in SQLPage front ends, and vice-versa.

13. Compiling SQLPage From Source

SQLPage binaries are available for most common server operating systems. If you need it to run on another OS, the compile process is simple. SQLPage is written in Rust, which has an easy to use tool chain, and many platform targets. Information about cross-compiling is at https://rust-lang.github.io/rustup/cross-compilation.html.

It's often easier to install Rust on your target device, and compile there (instead of cross-compiling). For example, the following steps can be used to compile SQLPage directly on Android in Termux (this has been tested on Android 13, using the v0.28.0 source files directly from Github):

```
pkg install rust
pkg install git
git clone https://github.com/lovasoa/SQLpage.git
cd SQLpage
cargo build --release
cd target/release
./sqlpage
```

A full printout of that console interaction is available at https://com-pute.com/nick/compile-sqlpage-in-termux-all-details.txt. It doesn't get much easier than that - even if you've never touched the Rust toolchain.

Here are 2 compiled executables that run in Android/Termux & in 32 bit Windows 8 (the 32 bit executable also runs in newer Windows versions, tested up to Windows 11):

https://com-pute.com/nick/sqlpageandroid
https://com-pute.com/nick/sqlpage32.exe

14. Hosting SQLPage Apps

14.1 Serving to a local network

You can deliver SQLPage apps over a local network simply by running SQLPage on a WIFI-connected computer (Window, Mac, Linux, Raspberry Pi, etc.). Just copy your .sql, .handlebars, .json and any other code/config files to the appropriate subdirectories where sqlpage.exe/sqlpage.bin is located. Be sure that the server computer is on and that the SQLPage server application is running, whenever the app needs to be accessible. If you're using any database system beside the embedded SQLite engine, make sure your database server software is also running and accessible. Configure your database connection string, port, and the folder from which .sql files are served, in /sqlpage/sqlpage.json. The example config file below is the one used to deliver all the live demo apps linked in this tutorial (adjust to your own preferences, or leave the default settings):

```
{
  "database_url": "sqlite://./sqlpage/sqlpage.db?mode=rwc",
  "listen_on":  "0.0.0.0:8008",
```

```
  "web_root": "./tutorial_text/sqlpage_tutorial_examples"
}
```

It's also possible to configure SQLPage by setting values in environment variables (see https://github.com/lovasoa/SQLpage/blob/main/configuration.md#configuring-sqlpage for instructions).

Client machines connected to the same local network can browse the IP address and port at which SQLPage is serving files (default port is 8080), plus the name of the .sql file to run (i.e., something like http://192.168.1.112:8080/myapp.sql). You can obtain the IP address of your server machine by running 'ipconfig' on the command line of that computer.

14.2 Using inexpensive hosted VPS

Inexpensive VPS accounts, hosted by companies such as A2hosting and Contabo, work basically the same way as serving apps locally. You can edit .sql files locally and copy them to the hosted server using scp, ftp, wget, or any other way you're comfortable transferring files. You can also create and edit .sql files directly with pico, vim, or any other editor, over an SHH connection, a remote desktop share, etc. If you need to connect to multiple databases for different apps, simply run multiple copies of the tiny SQLPage server, each with connection strings set to the selected databases. When connecting to a Linux server command line with SSH, use 'tmux' to run multiple console sessions. Doing that enables you to always get back to the same command line session where the server was started, to stop and re-start the server. Set up additional sessions to create, edit and delete .sql files. That enables you to edit your application files live, without having to stop or re-start the server. Linux VPS accounts begin around $2.99/month for commercial hosting, so getting apps running in production can be trivially inexpensive and quick to set up.

14.3 Using Docker, cloud hosting and other options

SQLPage is available as a Docker image at https://hub.docker.com/r/lovasoa/sqlpage. You can find a docker compose yaml file at https://github.com/lovasoa/SQLpage/blob/main/docker-compose.yml. There's also a version specifically prepared for use on AWS Lambda at https://github.com/lovasoa/SQLpage/releases. Docker is useful for installing on ARM based Raspberry Pi machines. Other install methods include brew (useful for Mac OS), nix, scoop, and cargo.

14.4 Host with datapage.app

To completely avoid any challenges involved with hosting SQLPage applications, go to https://datapage.app. They provide a turnkey solution to get SQLPage apps running online immediately - just upload .sql, .handlebars, and config files with FTP, and your apps are immediately working in production.

15. More Examples and Detailed Explanations

See http://learnsqlpage.com for more examples and more detailed explanations of the SQLPage code found in this text.

If you'd like a software product professionally developed, or to hire the author for tutoring, please call or text 215-630-6759, or send a message to nick@com-pute.com

COM-PUTE.COM