Practical Tips for Better PostgreSQL Applications

Marc Balmer <marc@msys.ch>

pgconf.de 2013
Oberhausen, Germany

Topics

1. Introduction
2. User Experience
3. Security
4. Colophon

1 Introduction

2 User Experience

3 Security

4 Colophon

About this presentation

- Aimed at application developers...
- ...who do client programming using libpq (or a wrapper around libpq for a language other than C)
- ...who know a little about PL/pgSQL
- ...who want to use PostgreSQL features
- demo code will be presented in the Lua language

About this presentation

- Aimed at application developers...
- ...who do client programming using libpq (or a wrapper around libpq for a language other than C)
- ...who know a little about PL/pgSQL
- ...who want to use PostgreSQL features
- demo code will be presented in the Lua language

About this presentation

- Aimed at application developers...
- ...who do client programming using libpq (or a wrapper around libpq for a language other than C)
- ...who know a little about PL/pgSQL
- ...who want to use PostgreSQL features
- demo code will be presented in the Lua language

About this presentation

- Aimed at application developers...
- ...who do client programming using libpq (or a wrapper around libpq for a language other than C)
- ...who know a little about PL/pgSQL
- ...who want to use PostgreSQL features
- demo code will be presented in the Lua language

About this presentation

- Aimed at application developers...
- ...who do client programming using libpq (or a wrapper around libpq for a language other than C)
- ...who know a little about PL/pgSQL
- ...who want to use PostgreSQL features
- demo code will be presented in the Lua language

What makes an application ,,Better”?

- Smaller code: Easier to maintain
- More robust: Immune to runtime problems (Network, etc.)
- More secure: Immune to SQL-Injection
- More responsive: Better user experience
- . . .

What makes an application ,,Better”?

- Smaller code: Easier to maintain
- More robust: Immune to runtime problems (Network, etc.)
- More secure: Immune to SQL-Injection
- More responsive: Better user experience
- . . .

What makes an application ,,Better”?

- Smaller code: Easier to maintain
- More robust: Immune to runtime problems (Network, etc.)
- More secure: Immune to SQL-Injection
- More responsive: Better user experience
- . . .

What makes an application „Better”?

- Smaller code: Easier to maintain
- More robust: Immune to runtime problems (Network, etc.)
- More secure: Immune to SQL-Injection
- More responsive: Better user experience
- . . .

What makes an application „Better”?

- Smaller code: Easier to maintain
- More robust: Immune to runtime problems (Network, etc.)
- More secure: Immune to SQL-Injection
- More responsive: Better user experience
- ...

The room capacity monitoring example

- The ,,capacity" table in the ,,pgconf" database:

Column	Type
room	character varying(32)
max_persons	integer
persons	integer

- When there are more than max_persons in a room, the room is over capacity. In the demo, max_persons of the ,,red" room is 30.

↪ Room capacity monitoring
↪ Room capacity monitoring with many clients

The room capacity monitoring example

- The ,,capacity" table in the ,,pgconf" database:

Column	Type
room	character varying(32)
max_persons	integer
persons	integer

- When there are more than max_persons in a room, the room is over capacity. In the demo, max_persons of the ,,red" room is 30.

↪ Room capacity monitoring
↪ Room capacity monitoring with many clients

The room capacity monitoring example

- The ,,capacity" table in the ,,pgconf" database:

Column	Type
room	character varying(32)
max_persons	integer
persons	integer

- When there are more than max_persons in a room, the room is over capacity. In the demo, max_persons of the ,,red" room is 30.

↪ Room capacity monitoring
↪ Room capacity monitoring with many clients

The room capacity monitoring example

- The ,,capacity" table in the ,,pgconf" database:

Column	Type
room	character varying(32)
max_persons	integer
persons	integer

- When there are more than max_persons in a room, the room is over capacity. In the demo, max_persons of the ,,red" room is 30.

↪ Room capacity monitoring

↪ Room capacity monitoring with many clients

The room capacity monitoring example

- The ,,capacity" table in the ,,pgconf" database:

Column	Type
room	character varying(32)
max_persons	integer
persons	integer

- When there are more than max_persons in a room, the room is over capacity. In the demo, max_persons of the ,,red" room is 30.

↪ Room capacity monitoring
↪ Room capacity monitoring with many clients

1 Introduction

2 User Experience

3 Security

4 Colophon

Asynchronous notifications: A PostgreSQL Feature

- Clients register for an event using **LISTEN** *name*
- Any client or the server can fire an event using **NOTIFY** *name*
- At any time a client can stop listening using **UNLISTEN** *name*

Asynchronous notifications: A PostgreSQL Feature

- Clients register for an event using **LISTEN** *name*
- Any client or the server can fire an event using **NOTIFY** *name*
- At any time a client can stop listening using **UNLISTEN** *name*

Asynchronous notifications: A PostgreSQL Feature

- Clients register for an event using **LISTEN** *name*
- Any client or the server can fire an event using **NOTIFY** *name*
- At any time a client can stop listening using **UNLISTEN** *name*

Asynchronous notifications to de-couple applications

- Trigger procedures can issue **NOTIFY**
- Use triggers e.g. on **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**
- Client executes **UPDATE**, server creates the notification

Asynchronous notifications to de-couple applications

- Trigger procedures can issue **NOTIFY**
- Use triggers e.g. on **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**
- Client executes **UPDATE**, server creates the notification

Asynchronous notifications to de-couple applications

- Trigger procedures can issue **NOTIFY**
- Use triggers e.g. on **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**
- Client executes **UPDATE**, server creates the notification

Process notifies after an SQL statement

```
conn:exec('listen capacity_changed_' .. room)
```

Process notifies after an SQL statement, cont'd.

```
conn:exec("update capacity set persons = "
    .. "persons + 1 where room = '" .. room .. "'")
local nam = 'capacity_changed_' ..room
local n = conn:notifies()
    while (n ~= nil) do
        if n:relname() == nam then
            res = conn:exec("select persons from "
                .. "capacity where room = '"
                .. room .. "'")
            textField:SetString(res:getvalue(1, 1))
        end
    n = conn:notifies()
end
```

↪ Process notifies after exec

Real-time behaviour

- Server sends notifies immediately on the socket
- „Watch" the connection socket: select(), XtAddInput(), GTK Input, etc.
- Process notifies when there is activity on the socket

Real-time behaviour

- Server sends notifies immediately on the socket
- ,,Watch" the connection socket: select(), XtAddInput(), GTK Input, etc.
- Process notifies when there is activity on the socket

Real-time behaviour

- Server sends notifies immediately on the socket
- ,,Watch" the connection socket: select(), XtAddInput(), GTK Input, etc.
- Process notifies when there is activity on the socket

Using XtAddInput

```
conn:exec("listen capacity_changed_" .. room)

inputId = app:AddInput(conn:socket(),
    processNotifies)
```

↪ Process notifies in real-time

Showing the connected clients

- The view pg_stat_activity shows connected clients
- We look at columns usename, client_addr, application_name
- We connect the usual way to the database:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost]])
```

↪ Show connected clients

Showing the connected clients

- The view pg_stat_activity shows connected clients
- We look at columns username, client_addr, application_name
- We connect the usual way to the database:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost]])
```

↪ Show connected clients

Showing the connected clients

- The view pg_stat_activity shows connected clients
- We look at columns usename, client_addr, application_name
- We connect the usual way to the database:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost]])
```

↪ Show connected clients

Showing the connected clients

- The view pg_stat_activity shows connected clients
- We look at columns usename, client_addr, application_name
- We connect the usual way to the database:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost]])
```

↪ Show connected clients

Showing the connected clients

- The view pg_stat_activity shows connected clients
- We look at columns usename, client_addr, application_name
- We connect the usual way to the database:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost]])
```

↪ Show connected clients

A better way to connect

An application name can be set when connecting:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost application_name=control(room_red)]])
```

application_name=control(room_red)

↪ Clients that set their names

A better way to connect

An application name can be set when connecting:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost application_name=control(room_red)]])
```

application_name=control(room_red)

↪ Clients that set their names

A better way to connect

An application name can be set when connecting:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost application_name=control(room_red)]])
```

application_name=control(room_red)

↪ Clients that set their names

A better way to connect

An application name can be set when connecting:

```
conn = pgsql.connectdb([[user=pgconf dbname=pgconf
    host=localhost application_name=control(room_red)]])
```

application_name=control(room_red)

↪ Clients that set their names

Unexpected behaviour

We still use the room capacity monitoring example

↪ Capacity monitoring

Unexpected behaviour

We still use the room capacity monitoring example

↪ Capacity monitoring

A server restart caused the application to misbehave

Connection setup:

```
conn = pgsql.connectdb(...)
if conn:status() ~= pgsql.CONNECTION_OK then
    print('Failed to connect to database')
    os.exit(1)
else
    updateRoom()
    conn:exec('listen capacity_changed')
    inputId = app:AddInput(conn:socket(),
        processNotifies)
end

app:MainLoop()
```

A closer look at the problem

- The application uses the X11 event loop
- It adds an X11 XtInput to the raw socket of the database connection to catch asynchronous notifications
- When the remote end closes the socket, this causes an endless loop
- A server restart causes the socket to be closed

A closer look at the problem

- The application uses the X11 event loop
- It adds an X11 XtInput to the raw socket of the database connection to catch asynchronous notifications
- When the remote end closes the socket, this causes an endless loop
- A server restart causes the socket to be closed

A closer look at the problem

- The application uses the X11 event loop
- It adds an X11 XtInput to the raw socket of the database connection to catch asynchronous notifications
- When the remote end closes the socket, this causes an endless loop
- A server restart causes the socket to be closed

A closer look at the problem

- The application uses the X11 event loop
- It adds an X11 XtInput to the raw socket of the database connection to catch asynchronous notifications
- When the remote end closes the socket, this causes an endless loop
- A server restart causes the socket to be closed

Connection setup, again

```
conn = pgsql.connectdb(...)
if conn:status() ~= pgsql.CONNECTION_OK then
    print('Failed to connect to database')
    os.exit(1)
else
    updateRoom()
    conn:exec('listen capacity_changed')
    inputId = app:AddInput(conn:socket(),
        processNotifies)
end

app:MainLoop()
```

processNotifies()

```
function processNotifies()
    conn:consumeInput()
    local n = conn:notifies()
    while (n ~= nil) do
        if n:relname() == 'capacity_changed' then
            updateRoom()
        end
        n = conn:notifies()
    end
end
```

The solution: Manage the connection

- Watch the connection status
- reset() the connection when needed

↪ Server restart, no harm done

The solution: Manage the connection

- Watch the connection status
- reset() the connection when needed

↪ Server restart, no harm done

The solution: Manage the connection

- Watch the connection status
- reset() the connection when needed

↪ Server restart, no harm done

Making the connection

```
conn = pgsql.connectdb(...)
if conn:status() ~= pgsql.CONNECTION_OK then
    textField:SetString('(Connecting to database)')
    app:AddTimeOut(1000, tryReconnectDatabase)
else
    updateRoom()
    conn:exec('listen capacity_changed')
    inputId = app:AddInput(conn:socket(),
        processNotifies)
end

app:MainLoop()
```

tryReconnectDatabase()

```
function tryReconnectDatabase()
    conn:reset()
    if conn:status() == pgsql.CONNECTION_BAD then
        app:AddTimeOut(1000, tryReconnectDatabase)
        return
    end
    updateRoom()
    inputId = app:AddInput(conn:socket(),
        processNotifies)
    conn:exec('listen capacity_changed')
end
```

processNotifies(), new version

```
function processNotifies()
    conn:consumeInput()
    if conn:status() == pgsql.CONNECTION_BAD then
        reconnectDatabase()
        return
    end
    local n = conn:notifies()
    while (n ~= nil) do
        if n:relname() == 'capacity_changed' then
            updateRoom()
        end
        n = conn:notifies()
    end
end
```

1 Introduction

2 User Experience

3 Security

4 Colophon

Security at the right layer

- Many applications handle security at the application layer, use only one database login
- **Surprise!** Software can have bugs. What if the application gets compromised?
- Full access to the application database by the intruder!

Security at the right layer

- Many applications handle security at the application layer, use only one database login
- **Surprise!** Software can have bugs. What if the application gets compromised?
- Full access to the application database by the intruder!

Security at the right layer

- Many applications handle security at the application layer, use only one database login
- **Surprise!** Software can have bugs. What if the application gets compromised?
- Full access to the application database by the intruder!

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Let the application ,,mirror" the security settings
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

↪ Webapplication security measures

Deny database administrator login

```
conn = pgsql.connectdb(...)
res conn:exec('select rolsuper from pg_roles '
    .. 'where rolname = current_user')

if res:ntuples() != 1 then
        -- No result, fail login
        os.exit(1)
end
if res:getvalue(1, 1) == 't' then
        -- db superuser, fail login
        os.exit(2)
end
```

Mirror user privileges in the application

```
res = conn:exec([[
SELECT groname FROM pg_group
    WHERE (
        SELECT usesysid FROM pg_user
        WHERE usename = current_user
    ) = ANY (grolist)
]]

for n = 1, res:ntuples() do
        -- use role membership to adjust UI
        -- has_role(res:getvalue(n, 1))
end
```

User input

An all time classic...

↪ Data entry application

User input

An all time classic...

↪ Data entry application

The code

```
function insertData()
    conn:exec(string.format([[
    INSERT INTO person (firstname, lastname, town)
    VALUES ('%s', '%s', '%s')
    ]],
    gui.entry.firstname:GetString(),
    gui.entry.lastname:GetString(),
    gui.entry.town:GetString()))
end
```

Good SQL

```
local a = 'Marc'
local b = 'Balmer'
local c = 'Basel'

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
a, b, c)
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', 'Basel')
```

Good SQL

```
local a = 'Marc'
local b = 'Balmer'
local c = 'Basel'

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
a, b, c)
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', 'Basel')
```

Malicious Input

```
'); truncate person; --
```

Malicious SQL

```
local a = 'Steve'
local b = 'B.'
local c = "'); truncate person; --"

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
a, b, c)
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Steve', 'B.', ''); truncate person; --')
```

Malicious SQL

```
local a = 'Steve'
local b = 'B.'
local c = "'); truncate person; --"

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
a, b, c)
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Steve', 'B.', ''); truncate person; --')
```

Solution 1/2: Escaping input

The same application, but this time the input is escaped

↪ Data entry application, with input escaping

Solution 1/2: Escaping input

The same application, but this time the input is escaped

↪ Data entry application, with input escaping

Inserting data with escaping

```
function insertData()
    conn:exec(string.format([[
    INSERT INTO person (firstname, lastname, town)
    VALUES ('%s', '%s', '%s')
    ]],
    conn:escape(gui.entry.firstname:GetString()),
    conn:escape(gui.entry.lastname:GetString()),
    conn:escape(gui.entry.town:GetString())))
end
```

Nice try, but...

```
local a = 'Steve'
local b = 'B.'
local c = '' '); truncate person; --"

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
conn:escape(a), conn:escape(b), conn:escape(c))
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Steve', 'B.', ''); truncate person; --')
```

Nice try, but...

```
local a = 'Steve'
local b = 'B.'
local c = "'); truncate person; --"

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
conn:escape(a), conn:escape(b), conn:escape(c))
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Steve', 'B.', ''); truncate person; --')
```

Solution 2/2: Using prepared statements

The same application, but a prepared statement is used

↪ Data entry application, with prepared statements

Solution 2/2: Using prepared statements

The same application, but a prepared statement is used

↪ Data entry application, with prepared statements

Inserting data with prepared statements, preparation step

```
function prepareConnection()
    conn:prepare('safe_entry', [[
    INSERT INTO person (firstname, lastname, town)
    VALUES ($1, $2, $3)
    ]], '', '', '')
end
```

Inserting data with prepared statements, execution step

```
function insertData()
    conn:execPrepared('safe_entry',
        gui.entry.firstname:GetString(),
        gui.entry.lastname:GetString(),
        gui.entry.town:GetString())
end
```

More advantages of prepared statements

- The statement is parsed only in the preparation step
- The query plan and optimizations are done in the preparation step

→ There is no setup time in the execution step

More advantages of prepared statements

- The statement is parsed only in the preparation step
- The query plan and optimizations are done in the preparation step

→ There is no setup time in the execution step

More advantages of prepared statements

- The statement is parsed only in the preparation step
- The query plan and optimizations are done in the preparation step

$\rightarrow$ There is no setup time in the execution step

More attack vectors

SQL must be composed carefully

- Whenever SQL is composed, extra care is needed
- *ALL* input must be sanitized
- Even when coming from sources we assume safe (Scanners etc.)
- Ever!
- Ever!

SQL must be composed carefully

- Whenever SQL is composed, extra care is needed
- *ALL* input must be sanitized
- Even when coming from sources we assume safe (Scanners etc.)
- Ever!
- Ever!

SQL must be composed carefully

- Whenever SQL is composed, extra care is needed
- *ALL* input must be sanitized
- Even when coming from sources we assume safe (Scanners etc.)
- Ever!
- Ever!

SQL must be composed carefully

- Whenever SQL is composed, extra care is needed
- *ALL* input must be sanitized
- Even when coming from sources we assume safe (Scanners etc.)
- Ever!
- Ever!

SQL must be composed carefully

- Whenever SQL is composed, extra care is needed
- *ALL* input must be sanitized
- Even when coming from sources we assume safe (Scanners etc.)
- Ever!
- Ever!

1 Introduction

2 User Experience

3 Security

4 Colophon

Questions?

Source code & Contact

The Lua interface to PostgreSQL
`https://github.com/mbalmer/luapgsql/`

Contact
Email: marc@msys.ch, mbalmer@NetBSD.org, m@x.org
Twitter: @mbalmer
IRC: mbalmer on freenode.net, #postgresql, #postgresql-de

`http://www.vnode.ch/`