2025-03-23, Chemnitz LinuxTage

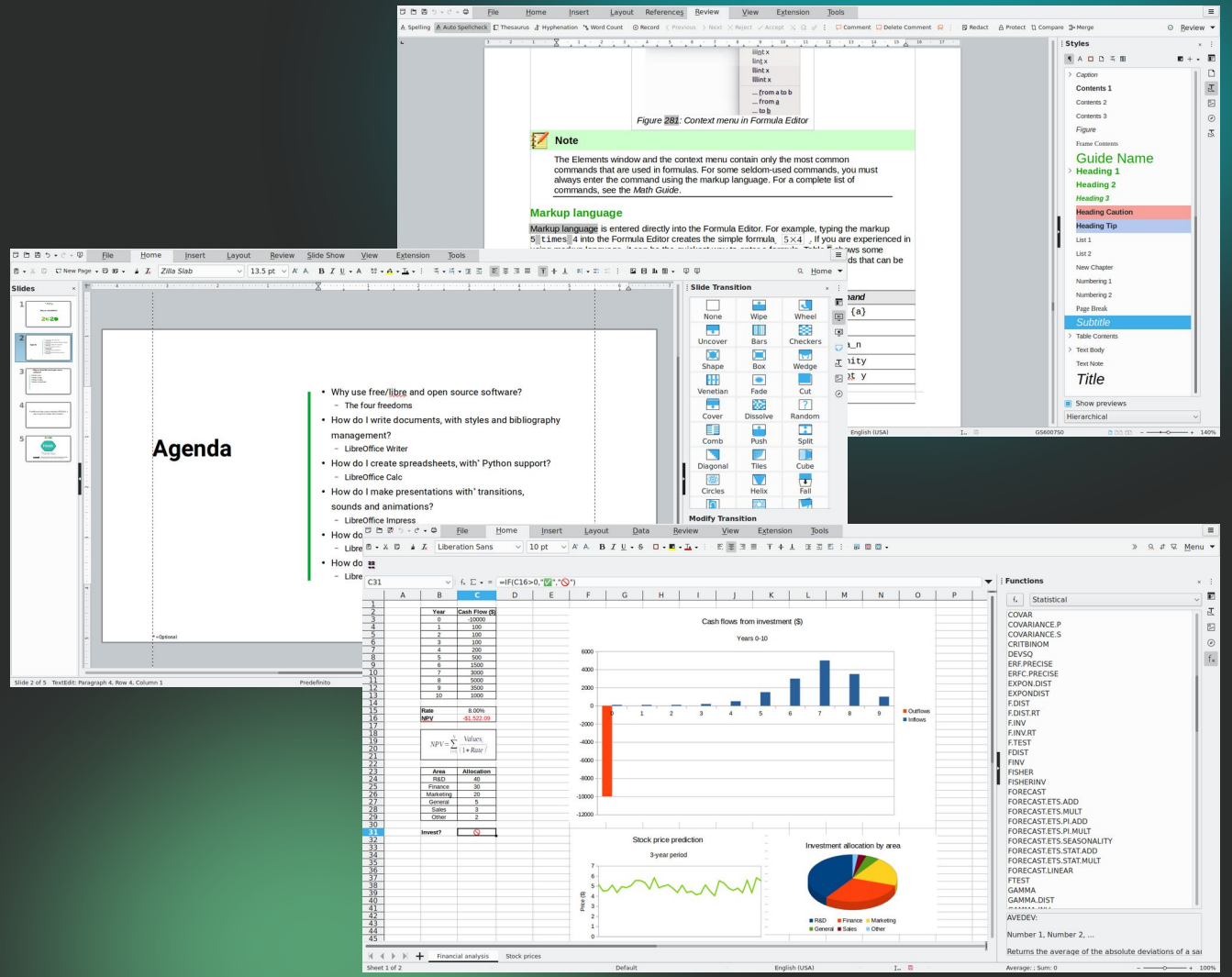
How to run LibreOffice inside your web page

WASM WebWidget & JavaScript

Thorsten Behrens <thorsten.behrens@allotropia.de>
allotropia software GmbH

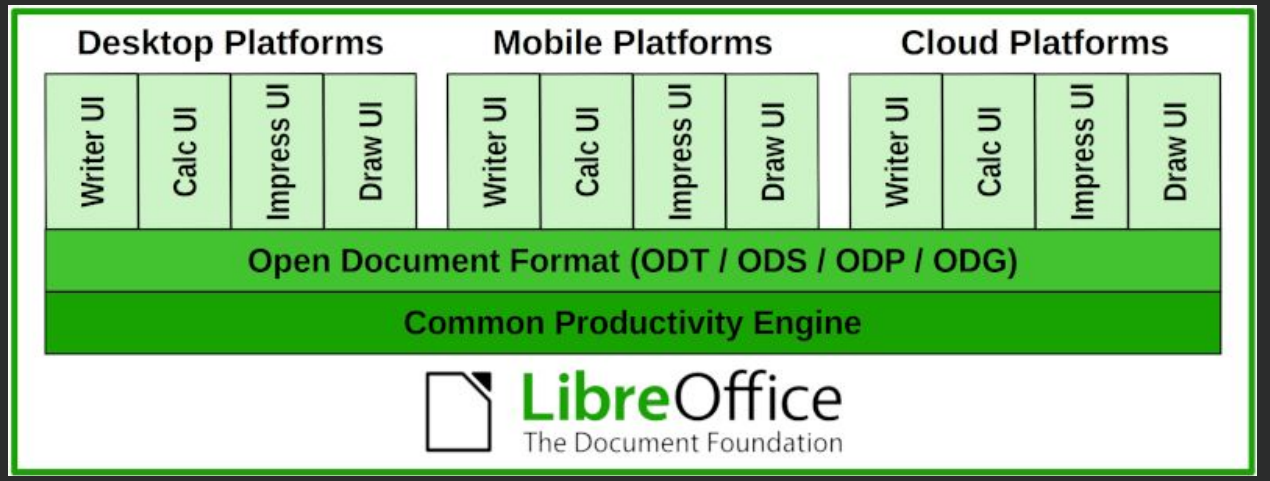
allotropia

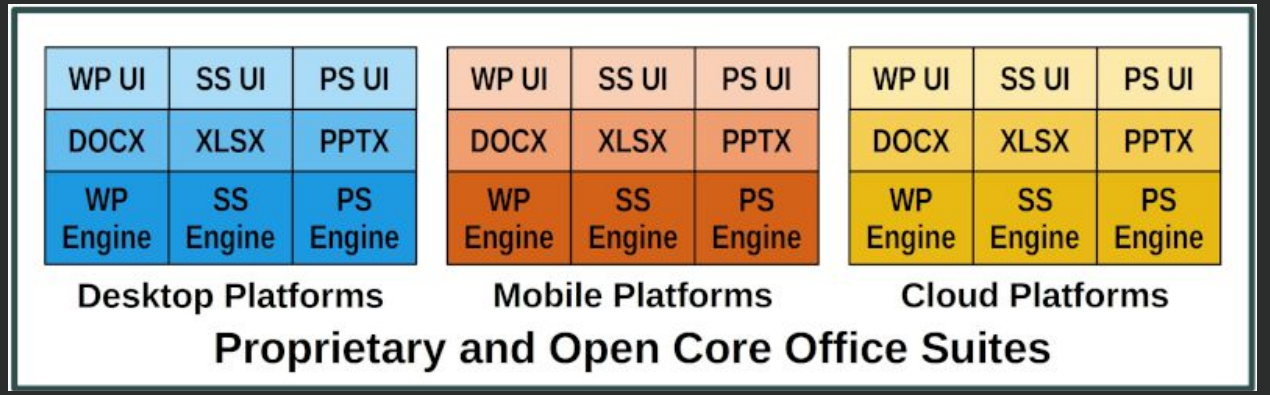
LibreOffice Technology



allotropia

LibreOffice Technology





allotropia

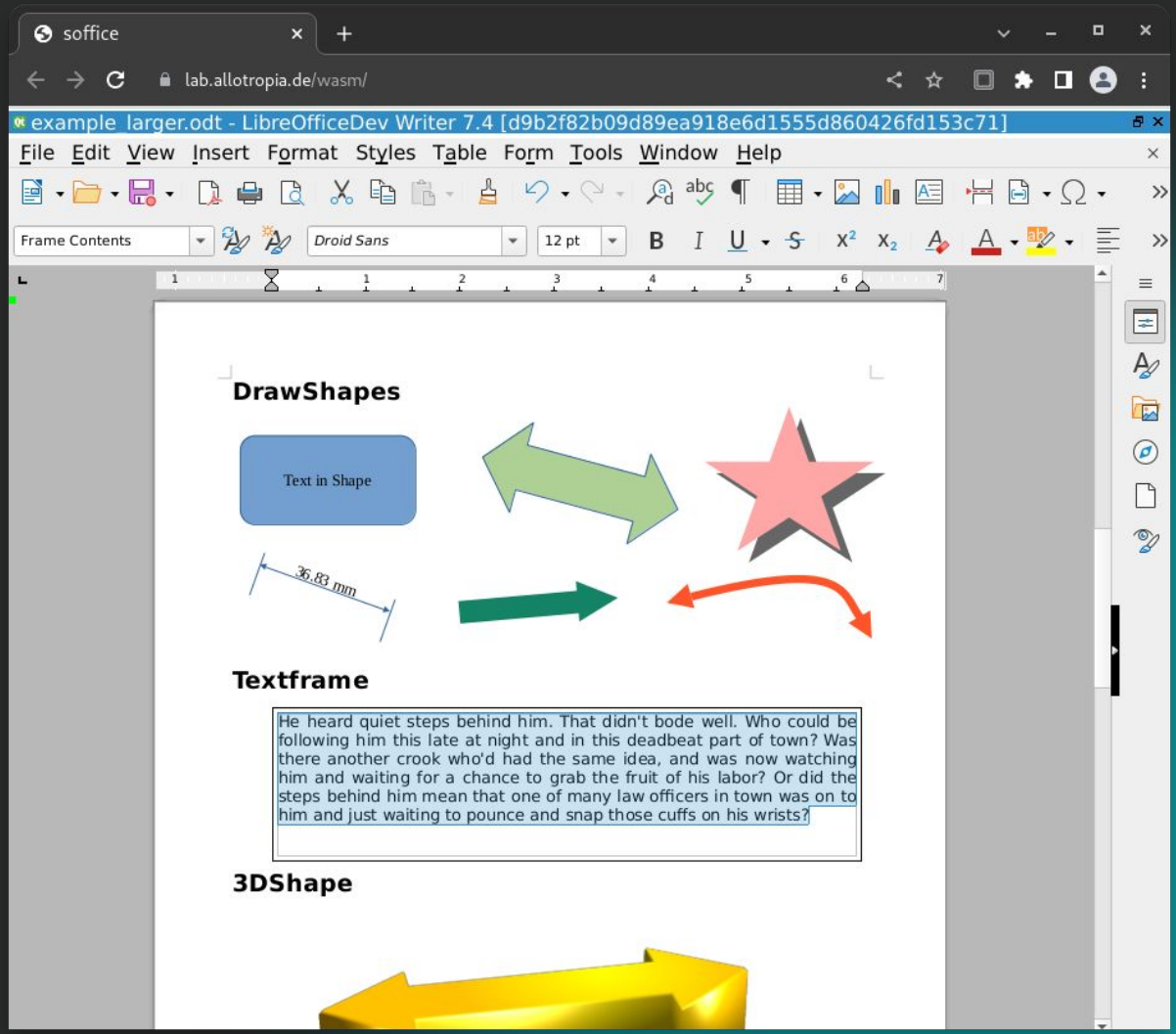
LibreOffice WASM

- LibreOffice in the browser
- fully client side – as a Web Assembly binary
- no server, no cloud services are needed
- built with the Emscripten toolchain
- currently using Qt for UI

allotropia

WA

LibreOffice WASM



allotropia

Timeline

Oct 2020
Project Kickoff w/ NLNet funding

Oct 2021
LibreOffice WASM pixel on the screen

LibreOffice WASM pixel on the screen
Oct 2021

Aug 2022
Calc supported

First LibreOffice WASM demo at FOSDEM
Feb 2022

April 2024
UNO API bindings for JavaScript

Headless PDF conversion
Feb 2022

ZetaJS Wrapper
Nov 2024

Size, speed & programmability improvements

Experimenting with frontends & toolkits

Timeline

- Project kickoff:
  - October 2020
- Build env & configury & emscripten setup
  - December 2020 – cross-building of a subset works; Docker builders for CI available
- Get the first LibreOffice-rendered pixel on the screen
  - October 2021 – after a death march of one year…
- Get Writer practically working
  - February 2022 – first fully working demo presented at FOSDEM

Timeline

- Get Calc and pdf export practically working:
  - August 2022 – got Calc working
  - September 2022 – got headless PDF conversion going
- Get Collabora Online port going:
  - January 2023 – got first demo working
- Got embind / UNO bindings going
  - April 2024: implementation end2end ready
  - first experiments & demo

allotropia

Timeline

- idiomatic JavaScript bindings
  - April `24 development started
  - October `24 launched v1.0 npm package
  - bugfixing, performance & size improvements
  - March `25 lots of papercuts solved
    - clipboard working, Impress support added, font antialiasing & canvas resizing

allotropia

Timeline

- Ongoing
  - bugfixing, performance & size improvements
  - further size reductions
  - OPFS support (WASM filesystem)
  - zetajs cleanups & convenience library

Step 1: WASM

Approach

- LibreOffice is an autotools & GNU make project
  - stuck with that, avoid other parallel build systems
  - and its already pretty portable, cross-compilation is supported out of the box
- LibreOffice has its own GUI abstraction
  - with plugins for Gtk, Qt/KF5, Win32 and OSX
  - with Qt5 supporting WASM natively, we went with that

Approach

- LibreOffice is basically c++ (by and large c++17)
  - we went with emscripten as platform compiler (pinned to 2.0.31 currently)
- We didn’t want to use any experimental WASM features
  - no threading
  - no dynamic linking (sadly require a re-tooling of the build system)
  - no native WASM exceptions
- We wanted to focus on Writer initially (and save size by not building/shipping the rest)

Approach

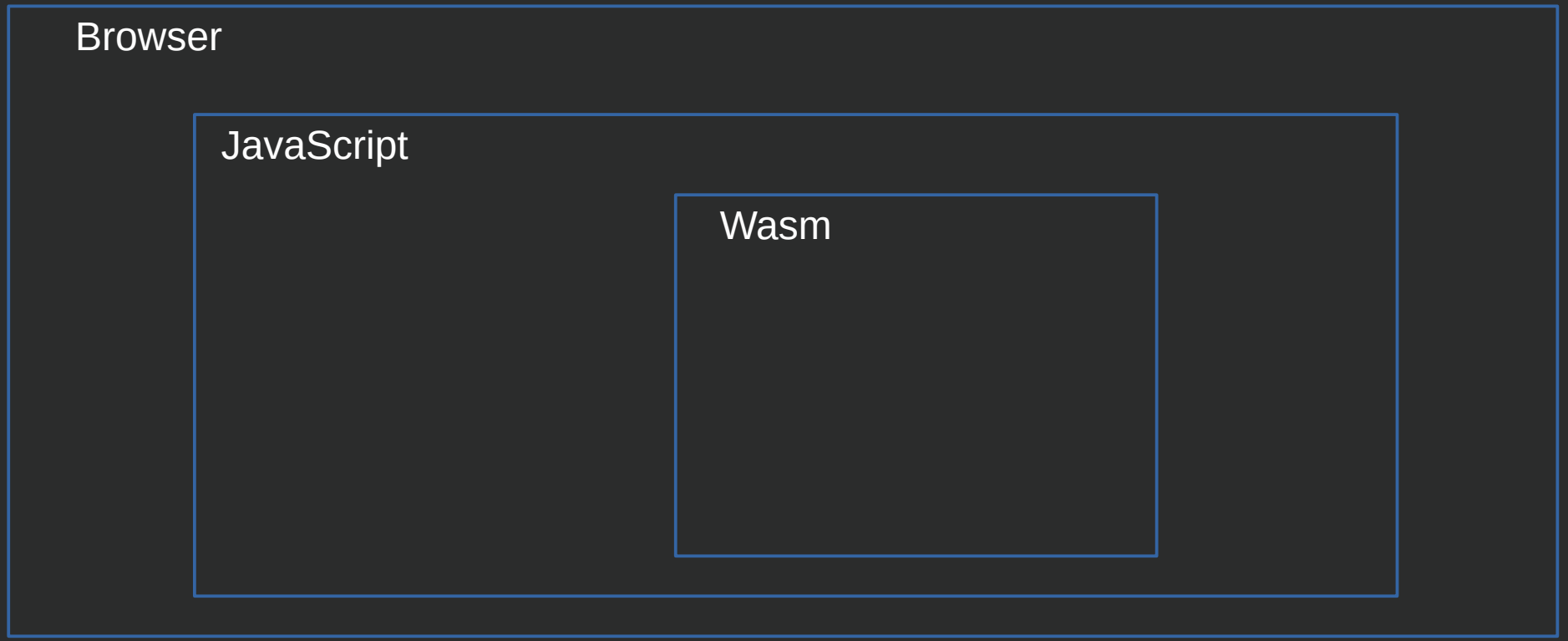
- emscripten & browser tools
  - several moving targets
  - random setups (`emsdk activate / install` not repeatable)
  - In 2020: no source-level debugging, SharedArrayBuffer limitations, unstable WASM impls
- LibreOffice gbuild make system w/o support for static linking
  - GNU make with a ton of `$(eval.. & $(call ..` self-made, functional build system
  - 88 commits, 4kLOC change to add that
- LibreOffice gbuild make system with dependency loops
  - UNO component system for dependeny inversion
  - once we link statically, we get loops

Approach

- LibreOffice UNO components
  - no static dependencies, but factory & runtime resolution
  - switched to static dependency per toplevel application
- LibreOffice needs a ton of secondary files (config, fonts, gui descriptions)
  - building a virtual embedded filesystem image
- Linker problems:
  - Link time grew quadratically with symbol amount
  - at some stage took >1h and >64GB to link
  - debug build now links in ~30s; not great but manageable
  - optimized build still needs huge amounts of memory and time, but saves 25% binary size with -O2
  - always separate debug data, downloadable on demand (DWARF)

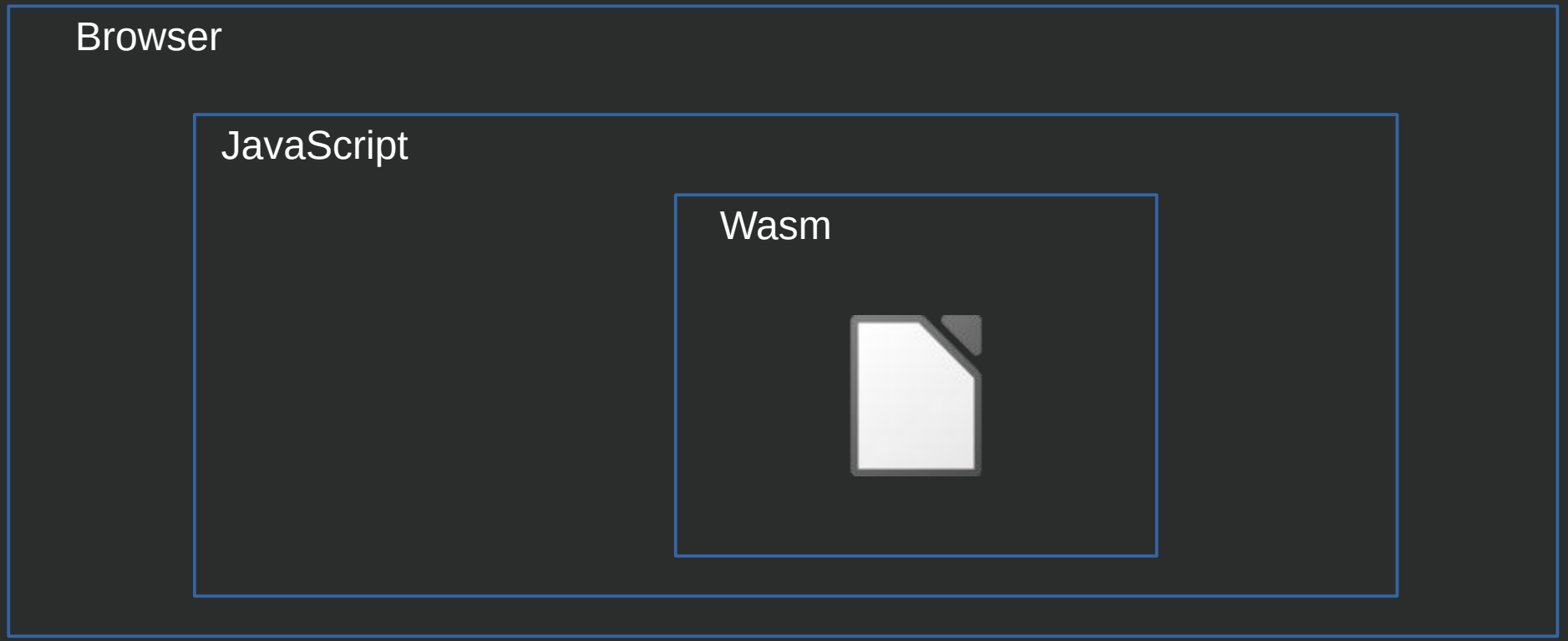
allotropia

Approach



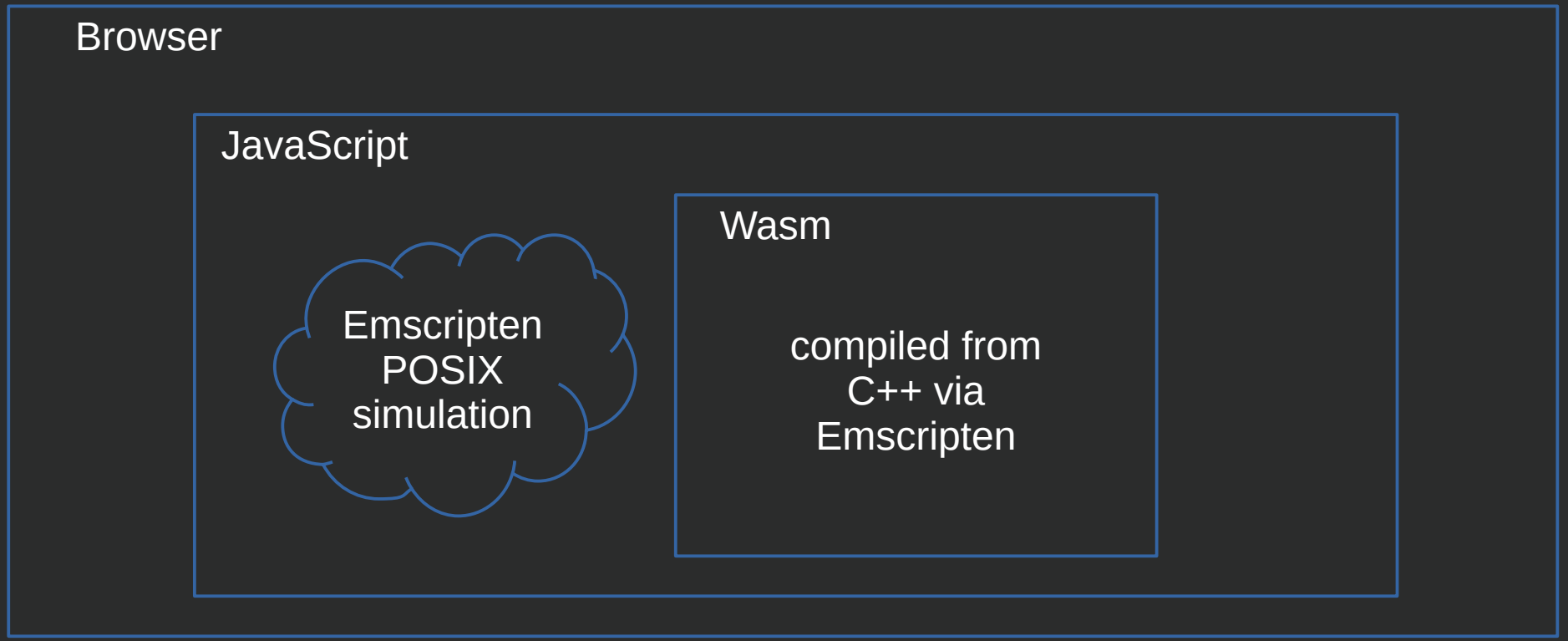
allotropia

Approach



allotropia

Approach



allotropia

Approach

Approach

- Every JS instance in the browser runs off an event loop:
  - onclick, oninput, ...
  - async/await, Promises
  - postMessage/onmessage

Approach

- LibreOffice runs off a main-thread event loop:
  - code handling one event
  - event loop
  - VCL/Qt
  - Application::Main
  - main
  - browser event loop

Approach

- LibreOffice runs off a main-thread event loop:
  - code handling one event
  - event loop
  - VCL/Qt
  - Application::Main
  - main
  - browser event loop

Approach

- turns out developing for WASM was super-hard
  - long link times, huge linking memory usage, impossible to get decent turn-around times, debugging WASM was not efficient - basically reading disassembled WASM code
    - → side-stepped via a native contraption
- turns out classical GUI application & JS event loops are incompatible
    - → side-stepped via threading
    - → waiting for `asyncify` to land
- be creative & ***never give up***! :)

LibreOffice WASM to ZetaJS

LibreOffice WASM in the browser

- ~working since end `23
- Qt5 for UI
- embedded in a canvas

UNO API exposed to JS

- ~working since early `24
- utilizes embind
- too verbose & lots of boilerplate
- not idiomatic JavaScript, using the C++ API via JS bindings

ZetaJS

- launched late `24
- idiomatic JS experience
- trivially embed on any webpage

allotropia

Step 2: embind

Approach

- LibreOffice has a rich (>4000 classes/types) programmability API
  - usable via Basic, Python, Java, C++, C##, etc
- needs to be available for a Web application
  - requires calling WASM code from JS
  - ...which is super-ugly – function ptrs, parameter mapping, return value mapping, ...

Approach

- emscripten’s `embind` to the rescue

```
#include <emscripten/bind.h>
using namespace emscripten;

float lerp(float a, float b, float t) {
    return (1 - t) * a + t * b;
}

EMSCRIPTEN_BINDINGS(my_module) {
    function("lerp", &lerp);
}
```

allotropia

Approach

- auto-generated for all of LibreOffice UNO
- available in the WASM binaries
- but: JS side of that is nasty
  - manual lifecycle handling
  - clunky lookup
  - no syntactic sugar, no automatic type conversions
  - extra-verbose (hyper-verbose at JS-side interfaces)

LibreOffice WASM to ZetaJS

LibreOffice WASM in the browser

- ~working since end `23
- Qt5 for UI
- embedded in a canvas

UNO API exposed to JS

- ~working since early `24
- utilizes embind
- too verbose & lots of boilerplate
- not idiomatic JavaScript, using the C++ API via JS bindings

ZetaJS

- launched late `24
- idiomatic JS experience
- trivially embed on any webpage

Step 3: zetaJS

allotropia

Approach

- Goals:
  - make using LibreOffice a pleasant & streamlined experience
- Via:
  - provide an `npm` package for 5-mins setup
  - have everything available via CDN – no need to deploy WASM binaries or mess with CORS headers
  - provide a nice, idiomatic JS wrapper: zetaJS

Approach

- zetaJS can:
  - map the UNO type system nicely into JS
  - like integers, strings, sequence, Any objects
  - UNO interface types & exceptions: mapped to JS types, throwing possible via `Module.throwUnoException`
  - provides iterators where useful
  - handles object lifecycles transparently

LibreOffice WASM to ZetaJS

LibreOffice WASM in the browser

- ~working since end `23
- Qt5 for UI
- embedded in a canvas

UNO API exposed to JS

- ~working since early `24
- utilizes embind
- too verbose & lots of boilerplate
- not idiomatic JavaScript, using the C++ API via JS bindings

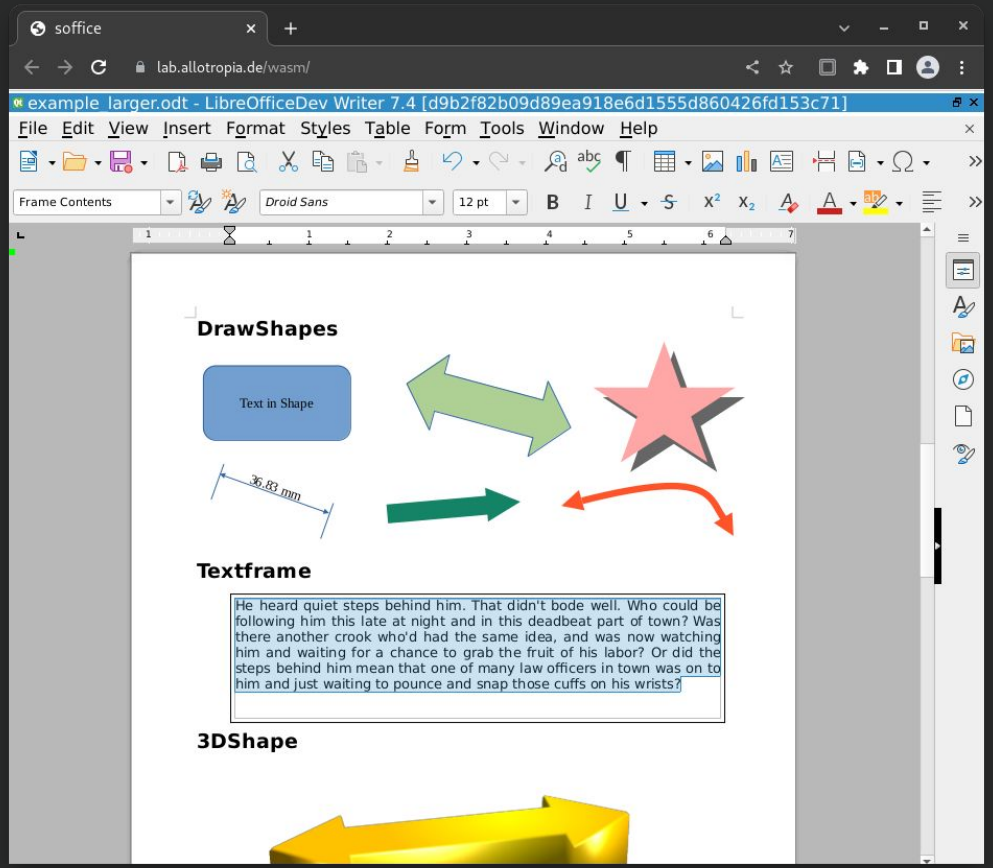
ZetaJS

- launched late `24
- idiomatic JS experience
- trivially embed on any webpage

Demo 1

https://zetaoffice.net/demos/web-office/

allotropia



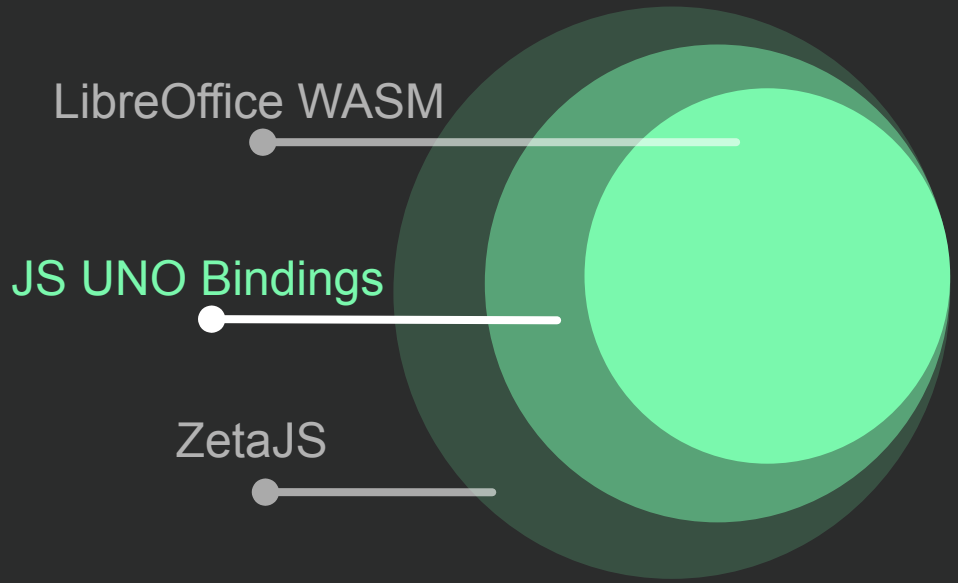
LibreOffice WASM

JS UNO Bindings

ZetaJS

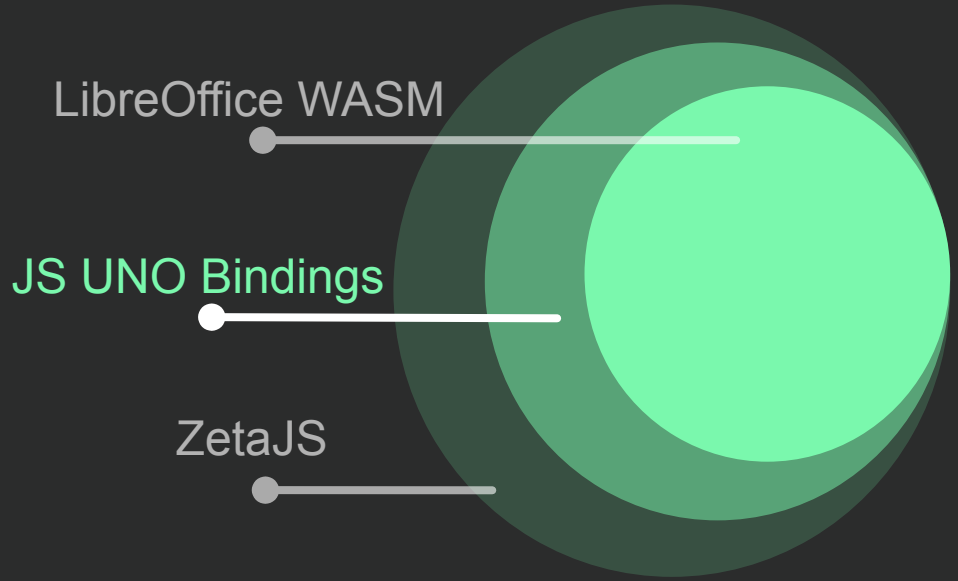
allotropia

```
const css = Module.uno.com.sun.star;
const desktop = css.frame.Desktop.create(
                  Module.getUnoComponentContext());
let xModel = desktop.getCurrentFrame().getController().getModel();
if (xModel === null || !css.text.XTextDocument.query(xModel))
{
    const args =
        new Module.uno_Sequence_com$sun$star$beans$PropertyValue(
        0, Module.uno_Sequence.FromSize);
    xModel = css.frame.XComponentLoader.query(desktop)
             .loadComponentFromURL(
               'file:///android/default-document/example.odt',
               '_default', 0, args);
    args.delete();
};
```

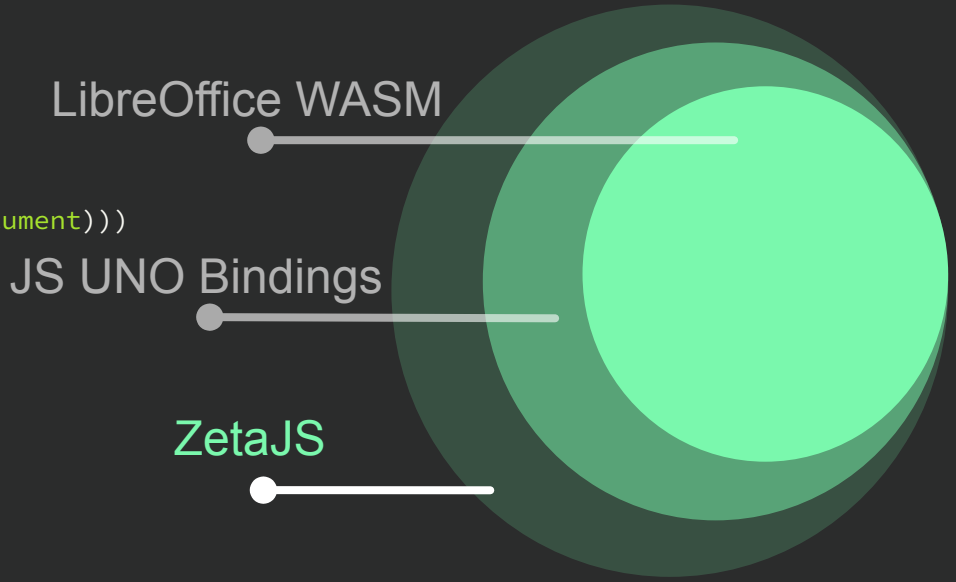


allotropia

```
const css = Module.uno.com.sun.star;
const desktop = css.frame.Desktop.create(
                  Module.getUnoComponentContext());
let xModel = desktop.getCurrentFrame().getController().getModel();
if (xModel === null || !css.text.XTextDocument.query(xModel))
{
    const args =
        new Module.uno_Sequence_com$sun$star$beans$PropertyValue(
        0, Module.uno_Sequence.FromSize);
    xModel = css.frame.XComponentLoader.query(desktop)
             .loadComponentFromURL(
               'file:///android/default-document/example.odt',
               '_default', 0, args);
    args.delete();
};
```




```
const css = zetajs.uno.com.sun.star;
const desktop = css.frame.Desktop.create(
                  zetajs.getUnoComponentContext());
let xModel = desktop.getCurrentFrame().getController().getModel();
if (!xModel?.queryInterface(zetajs.type.interface(css.text.XTextDocument)))
{
    xModel = desktop.loadComponentFromURL(
        'file:///android/default-document/example.odt',
        '_default', 0, []);
}
```



Why?

Data Privacy

Convenience

Extensibility

Scalability & Speed

Data Privacy | Convenience | Extensibility | Scalability & Speed

- Runs client sided in the browser, your data never leaves your machine.
- No server backend required for computing anything.
- Only ~50 MB WASM binary needs serving.

Data Privacy | **Convenience** | Extensibility | Scalability & Speed

User

- Web browser as the platform
- accessible from any device
- no installation, configuration ready to go – everywhere

Dev.

- w/ zetaJS trivially embeddable
- UNO API boilerplate abstracted away
- CDN at your disposal

(* but obviously you're not depended on it..)

Data Privacy

Convenience

Extensibility

Scalability & Speed

- Use UNO API directly from JavaScript
- Customize UI… or anything LibreOffice!
- Communicate both ways WASM ↔ JavaScript
- Implement any custom workflow!

allotropia

Data Privacy | Convenience | Extensibility | Scalability & Speed

- Serve it with ease
- No user count bottleneck. The client computes ;)
- No interaction latency
- Import/Export files immediately

More Demo time! :)

allotropia

Use cases

Example Use Cases

Integrate with Line of Business Applications

- order administration
- warehouse software
- accounting / bookkeeping
- document generation & conversion

...

Example Use Cases

Programmable documents Enable

- In-house mapping of processes
- Tons of powerful features

  macros, charts, letter templates, …

  spreadsheets, dynamic formatting, …

allotropia

How?

Challenges

Maturity of the platform

- Emscripten
- WASM
- Debugging...

Size of the resulting WASM binary

Currently: packed = 35M, optimised = 150M, debug = 200M + ~1G separate DWARF info

Size of the problem

Link time demands

How much code needs adapting?

Size of the filesystem image

~ 100M with all LO fonts, can be stored locally and split if needed

→ can use webfonts?

Recent News

Font aliasing issues squashed!

Browser scaling, high DPI support / improvements.

Impress is now supported.

Future Outlook

Experimenting with better UI toolkits

Typescript!

Better mobile/touch support

Better responsiveness

Give It A Spin!

See some more live demos
Visit <zetaoffice.net>

Use it in your favorite, front-end framework
Install via `npm i zetajs`

Check the examples
Visit <github.com/allotropia/zetajs/examples>

Questions & Answers