Express best practices

Agenda

- Discuss architecture
- Error handling pointers
- Performance pointers
- Security pointers

Architecture

Main concerns

- There are a few main concerns when building an Express app
  - Durability – error handling, graceful shutdown
  - Security – request sanitizing, SSL
  - Performance – compression, async code
  - Maintainability – code structuring, testing
- There many other concerns that should be addressed, could you suggest one?

Separation of concerns

- Try not to be naive when designing an app
- Separate network concerns & API declaration
- Use Express for it's fundamental http / web application features. That's it!
- Keep Express within its boundaries
  - Separate middleware and business logic
- Split the app into components
  - Will be discussed later on

Naive approach

- A common implementation of an express app mixes all the layers in one big horrible mess

```
app.get('/user/:id', async (req, res) => {
    try {
        const user = await DAL.getUserById(req); // returns User

        res.json(user.toJSON());
    } catch(e) {
        console.error('Failed to fetch user with error', e);

        res.status(500).send('Whoops, something went terribly wrong');
    }
});
```

Naive approach

- “Naive” implementation will lead to
  - Coupling with Express implementations
  - Boilerplate when writing tests
  - Lesser test coverage reports
  - A less maintainable codebase

Layering approach

- A more common hard headed approach will be to separate the app into component and then into layers
  - Router – web handler
  - Controller – mediation
  - Service – business logic
  - Model – data access
- Controller and service may be unified in smaller applications
- Can you think of the benefits?

Layering approach - PROS

- Decoupling from specific implementations
  - Better migration options (Koa, Hapi, Socket.io)
- Better testing options for each layer
- Can you think of other advantages?

Layering approach - CONS

- May lead to A LOT of boilerplate
  - Code spaghetti may be just around the corner
  - Duplication of code
  - More folders, more files -> more code to maintain
- Can you think of any other disadvantages?

Error handling

Error handling - general

- Always use a mature logger like Winston / Bunyan
  - Eliminate console.log / console.error from your code. It is synchronous!
- When in-doubt, gracefully restart
- Handle your code centrally, prevent handling code duplication
- Make sure to monitor with an APM tool

Error handling - Express

- Validate request input using a dedicated library
  - Joi will do the trick
- Avoid "on the spot" error handling
- Handle errors centrally
  - Reduces error handling code duplication
  - Express provides us with a middleware for error handling
- Distinguish between operational and internal errors

Error handling middleware

- Writing a naive error handling middleware is pretty straight forward

```
app.use(function errorHandler(err, req, res, next) {
    const error = "Huston, we have an error: " + err;

    logger.log('error', error);
    mailer.report().error('fatal', error);

    res.status(500);
    res.send('error', { error: err });
}
```

- Notice that the middleware accepts four arguments

Performance

Performance

- Use gzip to compress response body
- Do not block the loop, use async only functions
  - Use an async parsers to parse requests
  - Run your app with --trace-sync-io to print a warning every time it uses a sync API
- Delegate anything possible to a reverse proxy
  - Node is awful at doing CPU intensive tasks
  - Including gzip compression, SSL termination, throttling requests and static file serving

Performance

- Try and stay stateless, try and restart daily
- Monitor the heap - process.memoryUsage()
  - Javascript code has a tendency to leak
- Don’t forget to NODE_ENV=production

Security

Security

- Do not expose your errors
  - May reveal information about your service
- Only use secure cookies
- When in doubt, use a helmet (middleware)
  - Mitigates many common attack vectors
  - Really easy to implement
  - https://github.com/helmetjs/helmet

Let’s write some code

NODE.JS ARCHITECTURE

Agenda

- Discuss Node.js architecture
- Understand main characteristics
- Write some code

Characteristics

- Built on Chrome’s V8 engine
- Uses libuv
- Single threaded
- Event-driven
- Non blocking I/O

V8

- JavaScript engine
- Compiles JS to native machine code
- Written in C++
- Used in Chrome & Node.js, VSCode & Aton
- Supports Windows, macOS, Linux
- Can be embedded into C++

V8 vs. The World

- Same role as Java's JVM or .NET's CLR
- However, JavaScript is dynamic language
- Therefore less optimization opportunities
- V8 profiles code at runtime and optimizes it
  - Same as Java HotSpot technique
  - Has two compilers ignition & turbofan
  - Therefore can be faster than GCC
  - Shouldn't be faster than Java/.NET
  - See some benchmarks

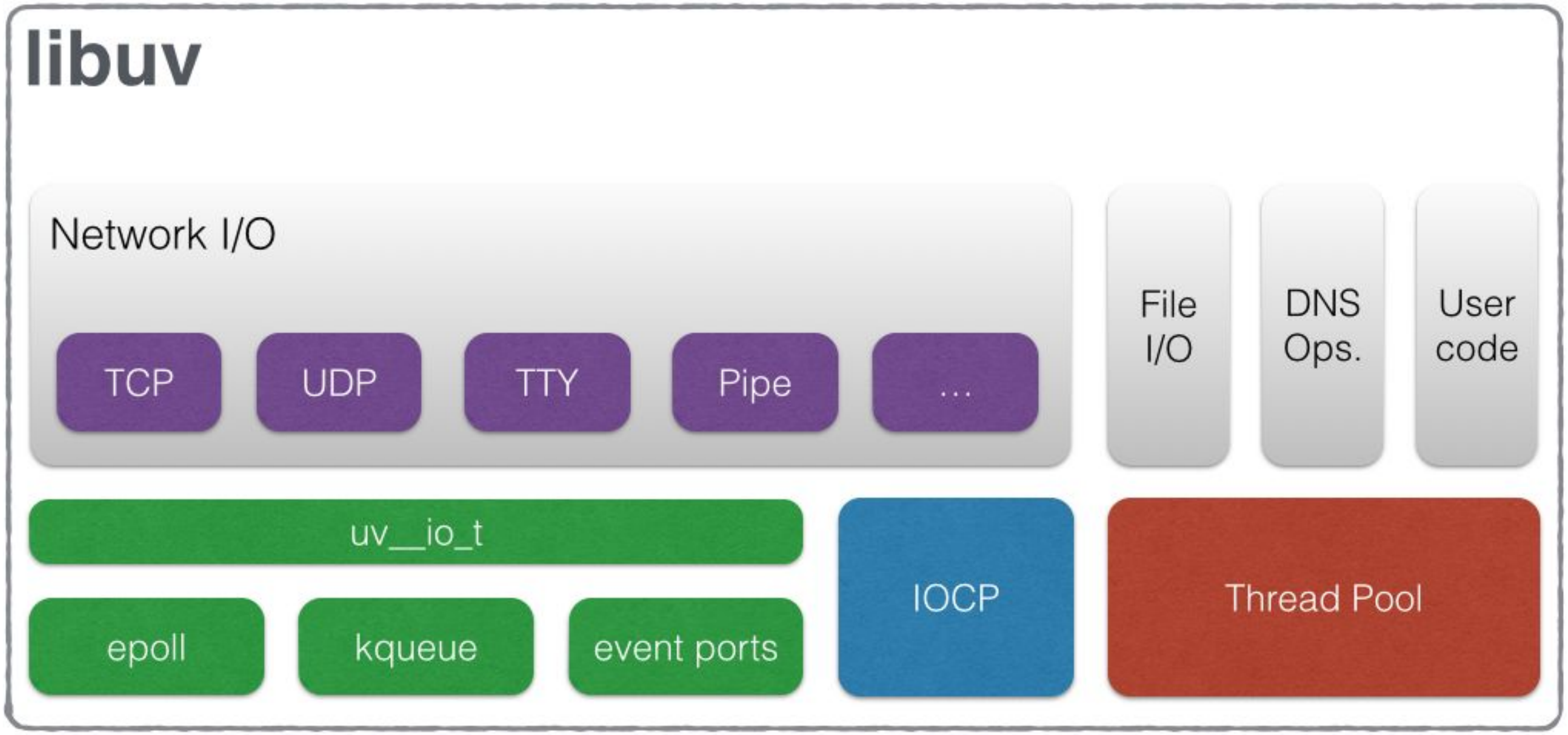
libuv

- Multi platform library with focus on asynchronous I/O
- Was developed for use by Node.js
  - But is now used by others
- Supports all the goodies of Node.js
  - Event loop
  - Async TCP & UDP sockets
  - Async file system operations
  - IPC
  - More …
- Create thread sample

libuv

- When possible uses OS asynchronous API
- Surprisingly does not use asynchronous file I/O
  - Code complexity
  - Poor APIs
  - Poor implementation
- Uses thread pool instead

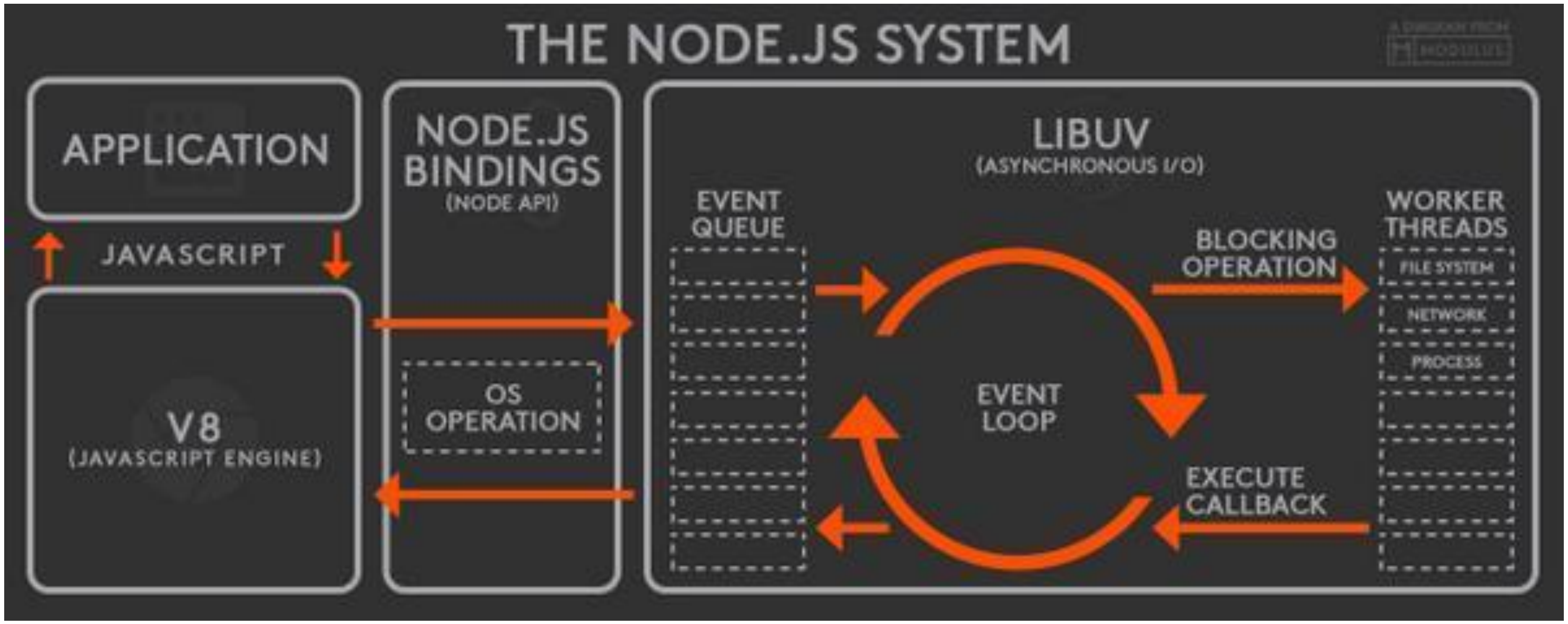
libuv


libuv
Network I/O
TCP
UDP
TTY
Pipe
...
File I/O
DNS Ops.
User code
uv__io_t
epoll
kqueue
event ports
IOCP
Thread Pool

Integrating

- Take V8
- Combine it with libuv
- Implement some JavaScript API to be consumed by the application
- And voila ... Node.js

Node.js Architecture


THE NODE.JS SYSTEM
A DIAGRAM FROM MODULUS
APPLICATION
JAVASCRIPT
V8
(JAVASCRIPT ENGINE)
NODE.JS BINDINGS
(NODE API)
OS OPERATION
LIBUV
(ASYNCHRONOUS I/O)
EVENT QUEUE
EVENT LOOP
BLOCKING OPERATION
WORKER THREADS
FILE SYSTEM
NETWORK
PROCESS
EXECUTE CALLBACK