
The Story of Kivik

Hi, I’m Jonathan

Go developer

Member CouchDB PMC

VP Engineering @ HUBUC

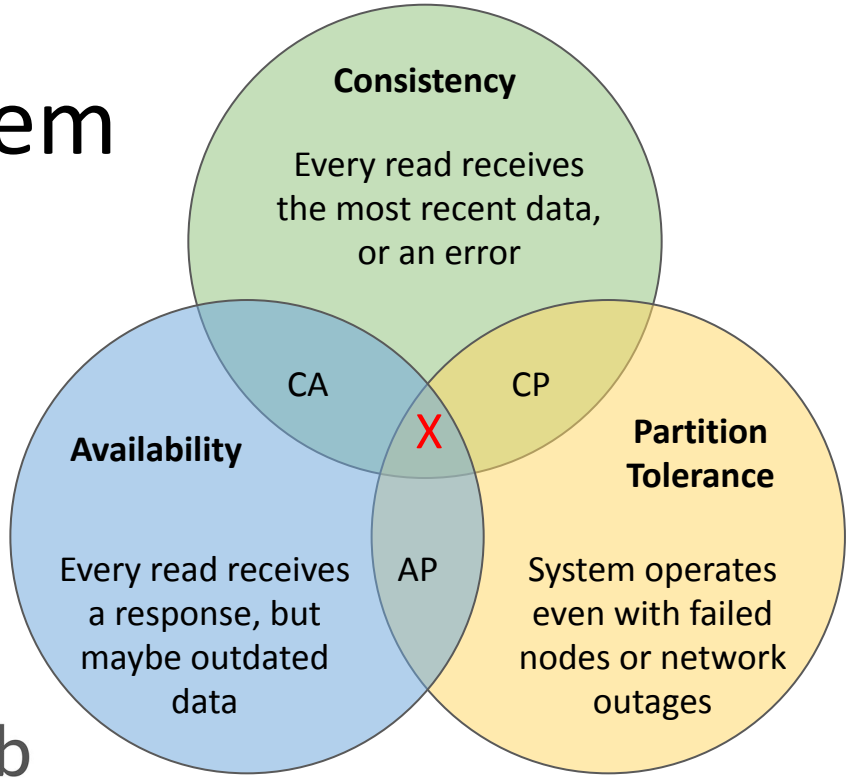
Agenda

- What is Kivik?
- What is CouchDB?
- Why another CouchDB driver?
- Challenges
- The Future


Kivik
A common interface to CouchDB and CouchDB-like databases for Go and GopherJS.

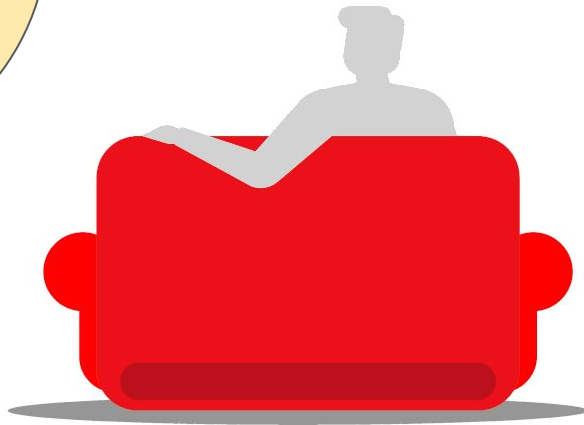

CouchDB
Seamless multi-master sync, that scales from Big Data to Mobile, with HTTP/JSON API and designed for Reliability.

CAP Theorem



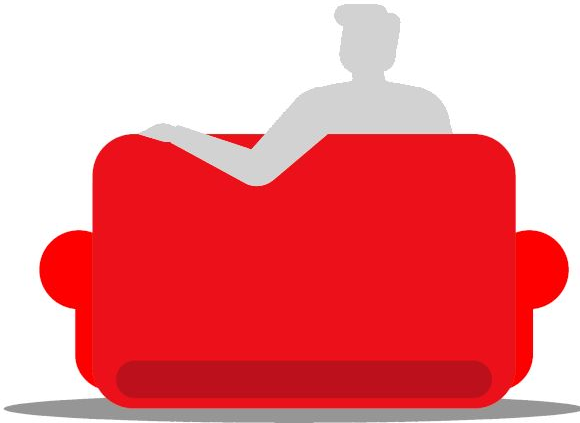
pouchdb

CouchDB
relax



The CouchDB API at a Glance

- CouchDB queries comprise HTTP verbs, specific endpoints, and an HTTP request
- CouchDB pretty strictly follows HTTP standards
  - Verbs: `GET`, `HEAD`, `PUT`, `POST`, `DELETE` (and `COPY`)
  - Request Headers: `Accept`, `Content-Type`, and a few others
  - Response Headers: `Content-Type`, `ETag`, and a few others
  - Status Codes: 200, 201, 202, 400, 404, etc
- Uses JSON
  - `application/json` for most requests and responses
  - Only rarely is anything else used



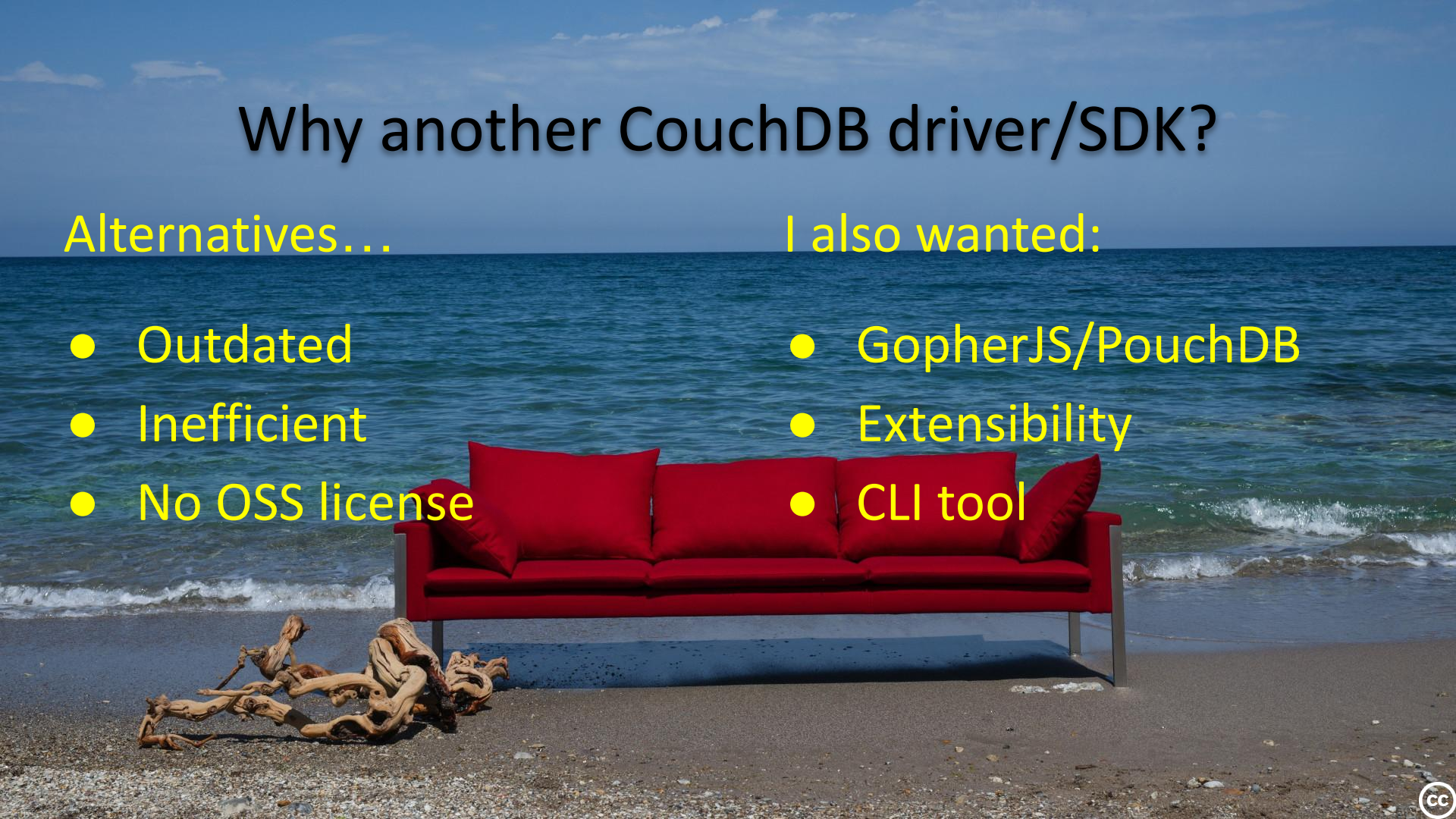
Why another CouchDB driver/SDK?

Alternatives…

- Outdated
- Inefficient
- No OSS license

I also wanted:

- GopherJS/PouchDB
- Extensibility
- CLI tool

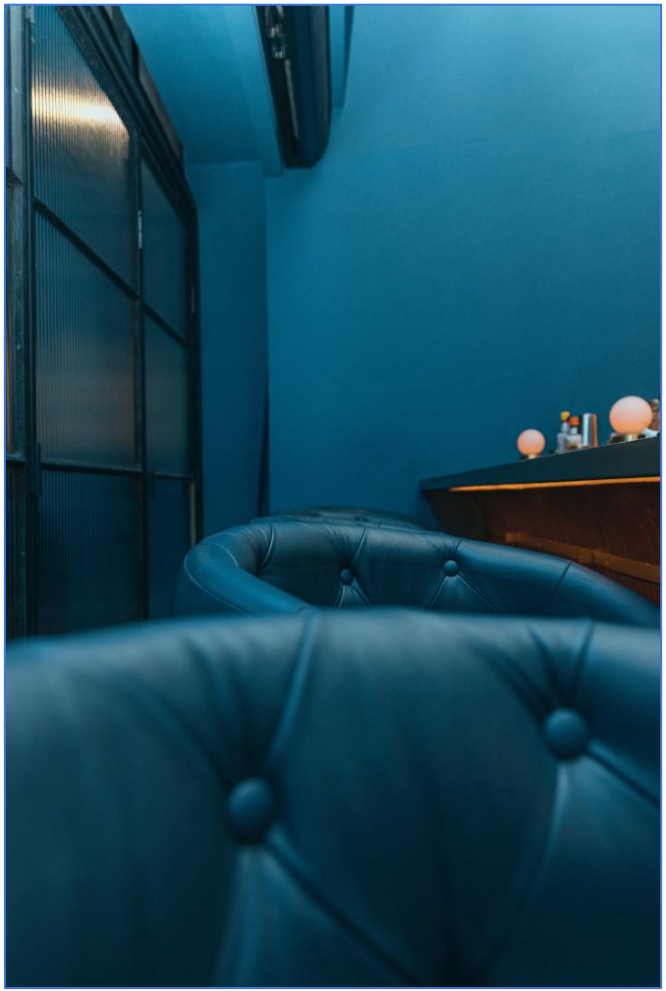


Design Goals

- JSON streaming support
- Open-source license
- Support for modern CouchDB features (i.e. `_find`)
- A single API for CouchDB & PouchDB & more
- An idiomatic Go interface, balanced with idiomatic CouchDB
- Easy to proxy some CouchDB requests through a Go server
- An extensible & composable suite of tools
- A unique name that’s easy to type and spell

My approach

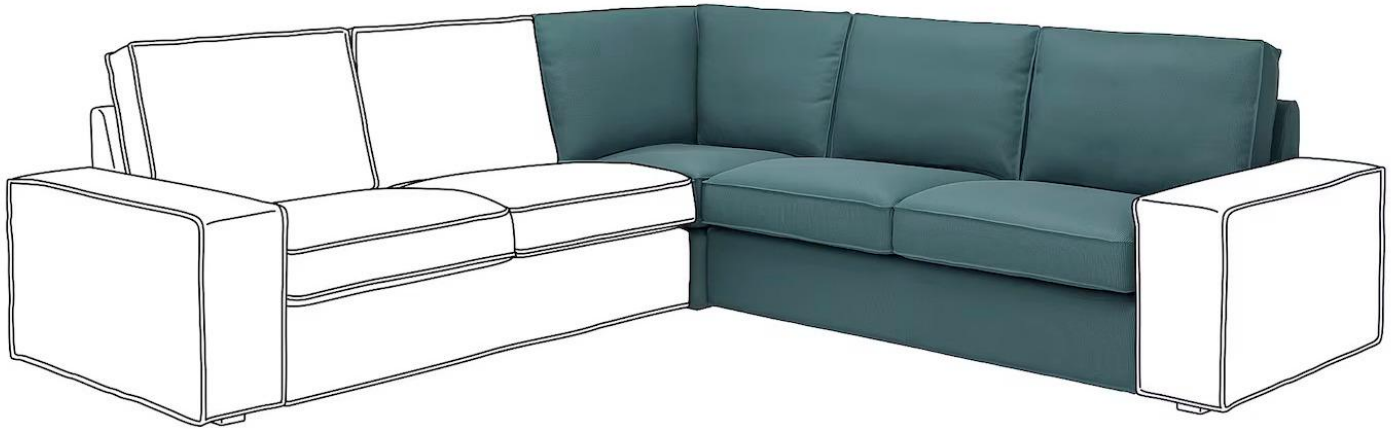
- A “driver” model inspired by `sql` & `sql/driver` in stdlib
- Multiple back-ends:
  - CouchDB
  - PouchDB
  - Mock
  - Filesystem
  - In-memory
  - Proxy
- Multiple front-ends:
  - Your own software
  - A CLI
  - A server



But what to call it?

- Something related to couches
- Something modular
- Pick and choose the parts you want

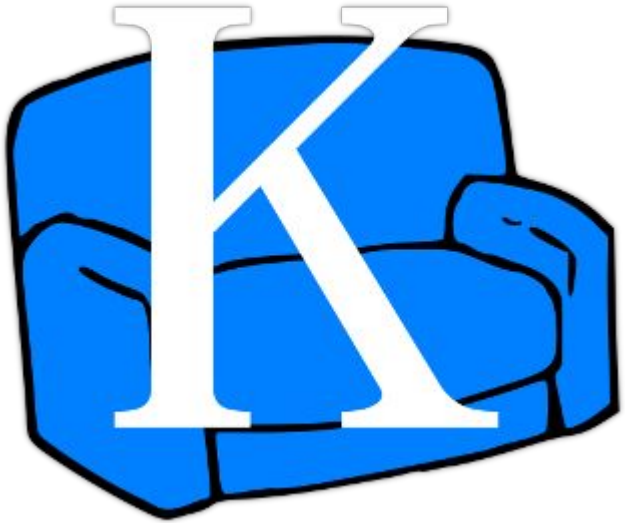
This business model sounds familiar…



Kivik at a Glance

`*kivik.Client` for most server-scoped requests:

```
GET /_all_dbs     ⇒ client.AllDBs()
POST /_replicate  ⇒ client.Replicate()
PUT /my_db        ⇒ client.CreateDB()
```

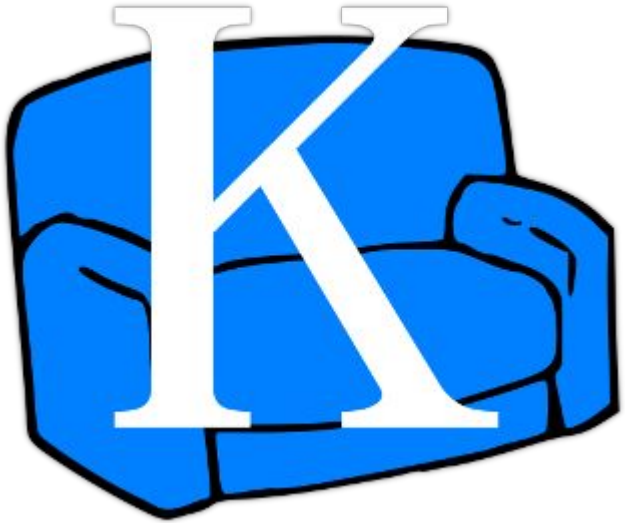


Kivik at a Glance

`*kivik.DB` for most database-scoped requests.

```
db := kivik.DB("my_db")

GET /my_db/_all_docs  ⇒ db.AllDocs()
POST /my_db           ⇒ db.CreateDoc()
PUT /my_db/doc1       ⇒ db.Put()
```




A Few Challenges
Error Handling
JSON Streaming
Re-parsing JSON

Sources of Errors

- Client errors (i.e. invalid URL)
- Driver errors (i.e. unsupported feature)
- Network errors (i.e. timeout)
- Server errors (i.e. document not found)

How can I support all of these in a consistent, and simple way?

Tried different approaches.

Current Approach

A single error type, and a helper function:

```
type Error struct {
      HTTPStatus int
      Message    string
      Err        error
}

func StatusCode(err error) int
```

And a few “fudged” status codes.

Checking an Error

```
err := db.Get(context.TODO(), "docID").ScanDoc(&doc)
if kivik.StatusCode(err) == http.StatusNotFound {
    return
}
```

Or use `errors.As`/`errors.Is` to check for specific network failures, etc.

JSON Streaming

- No stdlib support
- `json.Decoder` works, but is very cumbersome

```
func (i *iter) next(row interface{}) error {
	if i.dec == nil {
		i.dec = json.NewDecoder(i.body)
		if err := i.begin(); err != nil {
			return &kivik.Error{
				HTTPStatus: 502, Err: err}
		}
	}

	err := i.nextRow(row)
	if err != nil {
		if err == io.EOF {
			if e := i.finish(); e != nil {
				err = e
			}
			return err
		}
	}
	return err
}
```

```
func (i *iter) begin() error {
	// consume the first '{'
	if e := consume(i.dec, json.Delim('{')); e != nil {
		return e
	}
	for {
		key, err := nextKey(i.dec)
		if err != nil {
			return err
		}
		if key == i.expectedKey {
			// Consume the first '['
			return consume(i.dec, json.Delim('['))
		}
		if err := i.parseMeta(key); err != nil {
			return err
		}
	}
}
```

Re-Parsing JSON

Normally, the document revision is included in the ETag header of a response, making it easy to return without reading the entire response.

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 660
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT
ETag: "12-151bb8678d45aaa949ec3698ef1c7e78"

doc := db.Get(“my_doc”)
fmt.Println(doc.Rev)
// 12-151bb8678d45aaa949ec3698ef1c7e78
```

Re-Parsing JSON

However, in some cases that's not possible, as when returning the document with extra data (which would make an ETag-based cache invalid).

In such a case, it is necessary to parse the response body to extract the revision.

```
HTTP/1.1 200 OK
Content-Length: 660
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT

{"_id":"my_doc","_rev":"12-151bb8678d45aaa949ec3698ef1c7e78",...
```

```
func extractRev(resp *http.Response) (string, error) {
	buf := &bytes.Buffer{}
	r := io.TeeReader(resp.Body, buf)
	defer func() {
		// Restore the original resp.Body
		resp.Body = struct {
			io.Reader
			io.Closer
		}{
			Reader: io.MultiReader(buf, resp.Body),
			Closer: resp.Body,
		}
	}()
	rev, err := readRev(r)
	if err != nil {
		return "", err
	}
	return rev, nil
}
```

```
func readRev(r io.Reader) (string, error) {
	dec := json.NewDecoder(r)
	tk, err := dec.Token()
	if err != nil {
		return "", err
	}
	if tk != json.Delim('{') {
		return "", errors.New("wrong token")
	}
	for dec.More() {
		tk, err = dec.Token()
		if err != nil {
			return "", err
		}
		if tk == "_rev" {
			tk, err = dec.Token()
			if err != nil {
				return "", err
			}
			if value, ok := tk.(string); ok {
				return value, nil
			}
			return "", errors.New("invalid JSON")
		}
	}

	return "", errors.New("_rev key not found")
}
```

History

- June 2017: 1.0 release w/ CouchDB & PouchDB drivers

  Also partial/experimental support for:

  - Filesystem, in-memory, and proxy drivers
  - Server
- Feb 2020: 2.0/30 release w/ `kivikmock` for testing
- Dec 2020: CLI tool
  - Extensive filesystem support (without indexes)
  - Allows for easy disk-to-Couch or Couch-to-disk replication (analogous to pg_restore and pg_dump)

Future

- ???: 4.0 release (~75% complete)
  - Improved error handling
  - API improvements, particularly for multi-document responses
- Add support for full-text search, added in CouchDB 3.0
- Support incremental replication in the CLI tool (useful for incremental backup of a CouchDB server)
- Build Go indexing support, to allow indexing in the FS and in-memory backends
- Flesh out the server tool, possibly merge it with the CLI tool
- Rename package import path to `kivik.io`?
- Re-consolidate drivers into main package?


Questions?

Other Resources

CouchDB website: https://couchdb.apache.org/

Kivik website: http://kivik.io/

YouTube Video: Advanced JSON Handling in Go

Very incomplete eBook: Data Serialization in Go

New YouTube Channel: Boldly Go

Slides at:
https://jhall.io/kivik-story