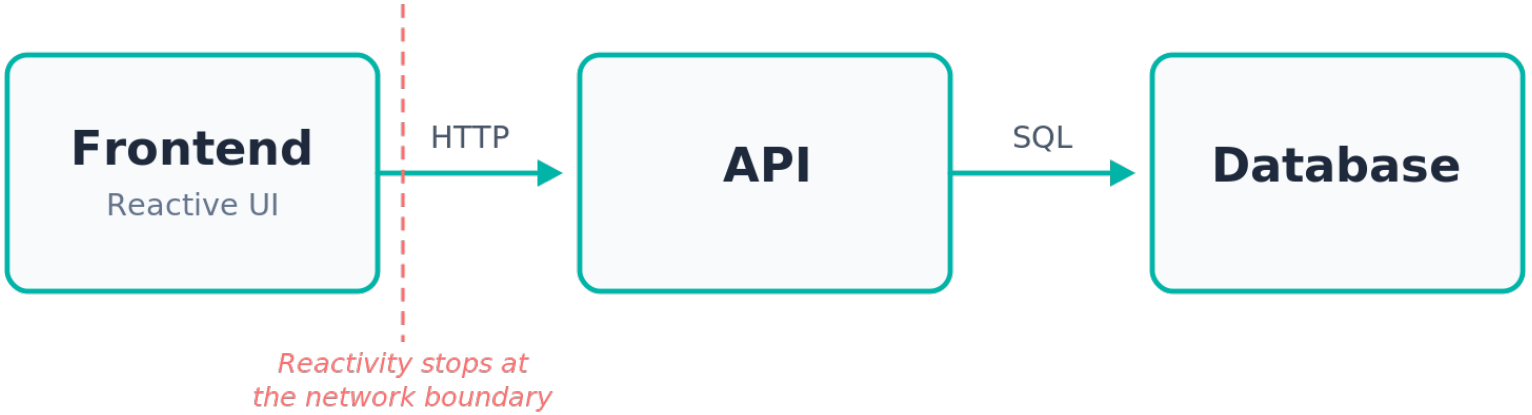
Query-driven Sync
with TanStack DB

Kevin De Porre • ElectricSQL

FOSDEM 2026

Traditional Web Architecture



Backend changes require polling, WebSockets, or SSE to reach the UI

Local-First Architecture

Local-First Architecture

✓ **Reads & writes feel instant — Great UX!**

✗ Rewrite frontend/backend around sync engine

✗ Initial sync is slow/unfeasible for big datasets

✗ Querying big local datasets is slow → UI becomes laggy

- Need indexing and optimized querying
- Build your own client-side DB? 🐰⛳

What Do We Actually Want?

Great UX

Fast apps, no loading spinners

→ Data must be locally available

Great DX

Declare what data you need

→ Not how (HTTP, caching, syncing...)

What Do We Actually Want?

Great UX

Fast apps, no loading spinners

→ Data must be locally available

Great DX

Declare what data you need

→ Not how (HTTP, caching, syncing...)

The Challenge:

How to sync data just-in-time so it's available

when the user needs it — without requiring dev to be a 🧙

What Do We Actually Want?

Great UX

Fast apps, no loading spinners

→ Data must be locally available

Great DX

Declare what data you need

→ Not how (HTTP, caching, syncing...)

The Challenge:

How to sync data just-in-time so it's available

when the user needs it — without requiring dev to be a 🧙

→ This led us to develop TanStack DB

What is TanStack DB?

Reactive Client-Side Store

Data updates flow automatically to your UI

Backend-Agnostic Collections

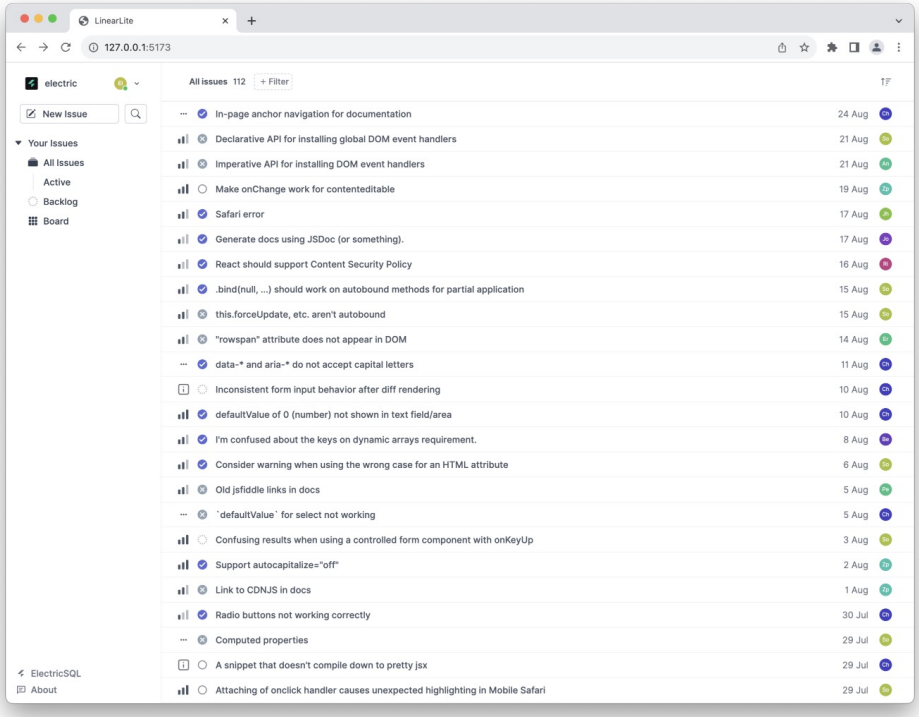
Populate from any data source — REST, GraphQL, sync engines

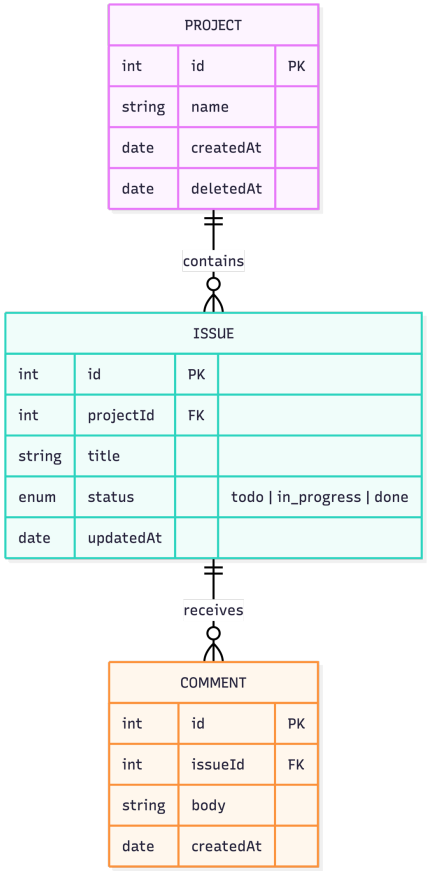
Incremental Queries

SQL-like queries that update efficiently on changes

Think of it as a client-side database with IVM (Incremental View Maintenance)

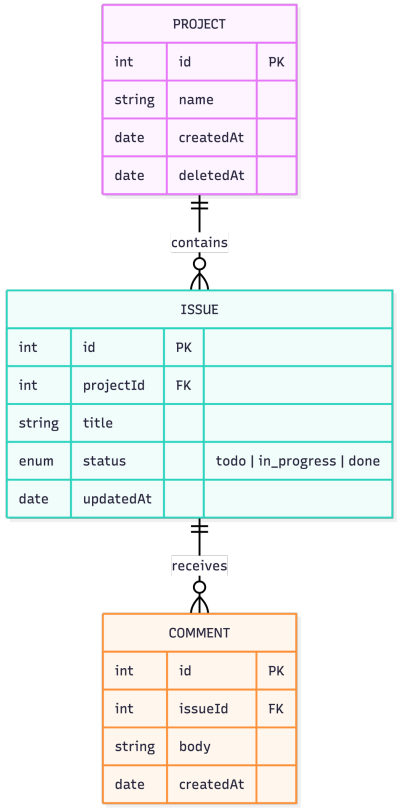
Example: Issue Tracker App





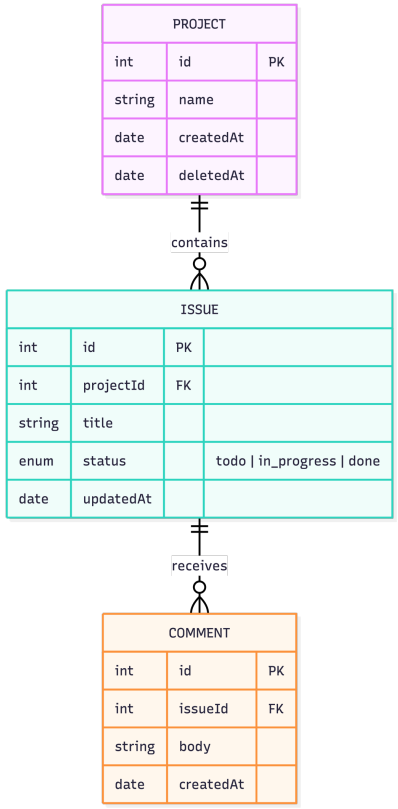
Issue Tracker: CRUD API

```
// api.ts - Standard fetch-based implementation

function getProjects(): Project[] {
  return fetch('/api/projects').json()
}

function getIssues(): Issue[] {
  return fetch('/api/issues').json()
}

function getComments(): Comment[] {
  return fetch('/api/comments').json()
}
```



Issue Tracker: CRUD API

```
// api.ts - Standard fetch-based implementation

async function createProject(input: CreateProjectInput) {
  const response = await fetch('/api/projects', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(input),
  })
  return response.json()
}
```



Data Collections

```
// collections.ts
const projectsCollection = createCollection(
  queryCollectionOptions({
    queryClient,
    queryKey: ['projects'],
    queryFn: api.getProjects,
    getKey: (p: Project) => p.id,
    onInsert: async (project) => {
      await api.createProject(project)
    },
    onUpdate: async (project) => { /* API call */ },
    onDelete: async (project) => { /* API call */ }
  }),
)
```

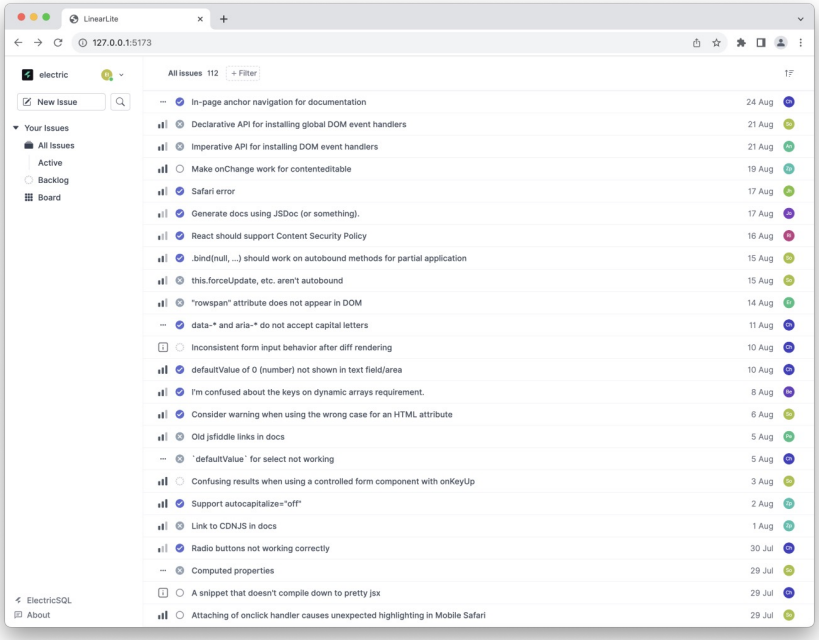
Data Collections

```
const issuesCollection = createCollection(
  queryCollectionOptions({
    queryClient,
    queryKey: ['issues'],
    getKey: (i: Issue) => i.id,
    queryFn: api.getIssues,
  }),
)

const commentsCollection = createCollection(
  queryCollectionOptions({
    queryClient,
    queryKey: ['comments'],
    getKey: (c: Comment) => c.id,
    queryFn: api.getComments,
  }),
)
```

Collections wrap existing API
no backend changes needed

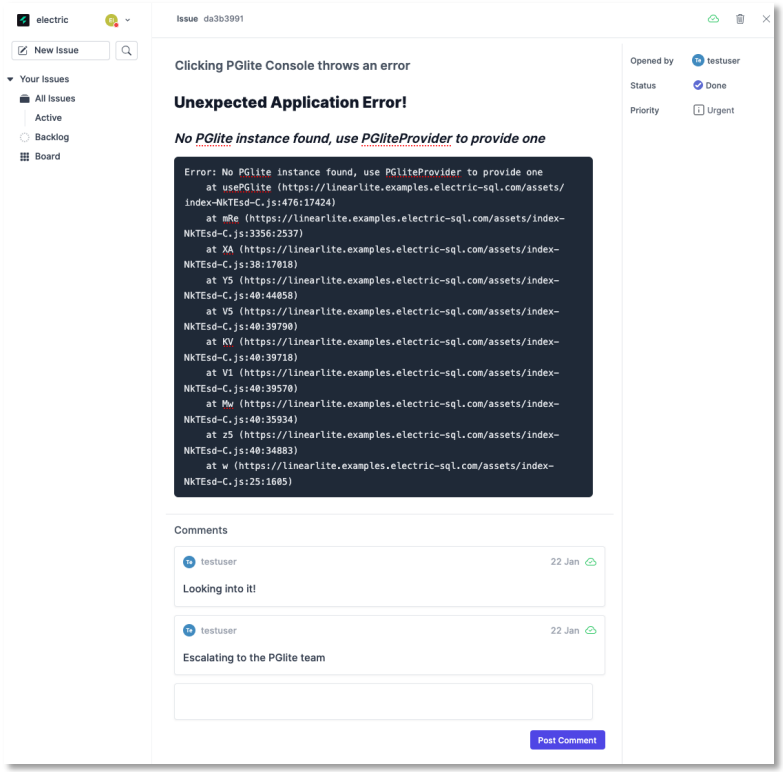
Issues View

```
function IssuesList({ projectId, status }) {
  const { data: issues } = useLiveQuery(
    (q) =>
      q.from({ i: issuesCollection })
       .where(({ i }) => eq(i.projectId, projectId))
       .where(({ i }) => eq(i.status, status))
       .orderBy(({ i }) => i.updatedAt, 'desc'),
    [projectId, status])

  if (isLoading) return <div>Loading...</div>

  return (
    <ul>
      {issues.map((i) => <li key={i.id}>{i.title}</li>)}
    </ul>
  )
}
```



Issue Detail View

```
function IssueDetail({ issueId }) {
  const { data: rows } = useLiveQuery(
    (q) => q.from({ i: issuesCollection })
      .where(({ i }) => eq(i.id, issueId))
      .leftJoin(
        { comment: commentsCollection },
        ({ i, comment }) => eq(i.id, comment.issueId))
      .orderBy(({ comment }) => comment.createdAt, 'asc'),
    [issueId]
  )

  if (isLoading) return <div>Loading...</div>
  const issue = rows[0]!.issue
  const comments = rows.map((r) => r.comment)
  // render issue + comments
}
```



- One declarative query
- Filters and joins use available indexes for efficiency

Adding Comments — Optimistic Writes

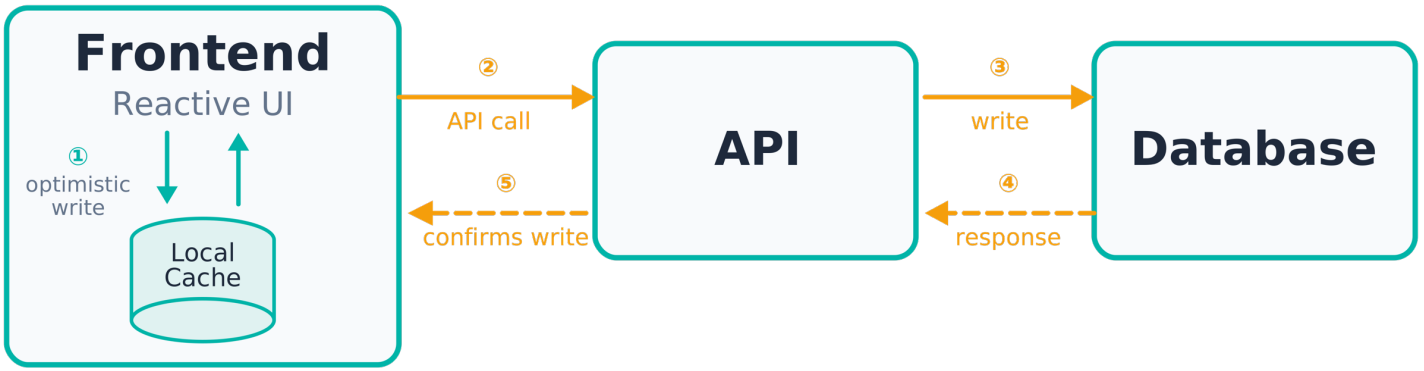
```
// Optimistic insert — UI updates immediately
const addComment =
  (issueId: string, text: string) =>
    commentsCollection.insert({
      id: crypto.randomUUID(),
      issueId,
      body: text,
      createdAt: Date.now(),
    })
```

Adding Comments — Optimistic Writes

```
// Optimistic insert — UI updates immediately
const addComment =
  (issueId: string, text: string) =>
    commentsCollection.insert({
      id: crypto.randomUUID(),
      issueId,
      body: text,
      createdAt: Date.now(),
    })
```

```
const projectsCollection = createCollection(
  queryCollectionOptions({
    onInsert: async (project) => {
      await api.createProject(project)
    },
    // ...
  }),
)
```

Adding Comments — Optimistic Writes



Optimistic update shows immediately; confirmed when API response arrives

Incremental View Maintenance (IVM)

```
q.from({ i: issuesCollection })
 .where(({ i }) => eq(i.id, issueId))
 .leftJoin(
   { comment: commentsCollection },
   ({ i, comment }) => eq(i.id, comment.issueId))
 .orderBy(({ comment }) => comment.createdAt, 'asc')
```

Incremental View Maintenance (IVM)

```
q.from({ i: issuesCollection })
 .where(({ i }) => eq(i.id, issueId))
 .leftJoin(
   { comment: commentsCollection },
   ({ i, comment }) => eq(i.id, comment.issueId))
 .orderBy(({ comment }) => comment.createdAt, 'asc')
```

Traditional: Re-run entire query on every change

→ O(n log n) for sorting, O(n × m) for joins

IVM: Incrementally update based on the change

→ O(log n) for sorted insert, O(max(n, m)) for incremental join

How query results update:

- Don't recompute entire query
- Output delta changes
- E.g. binary search for sorted insert

Incremental View Maintenance (IVM)

```
q.from({ i: issuesCollection })
 .where(({ i }) => eq(i.id, issueId))
 .leftJoin(
   { comment: commentsCollection },
   ({ i, comment }) => eq(i.id, comment.issueId))
 .orderBy(({ comment }) => comment.createdAt, 'asc')
```

How query results update:

- Don't recompute entire query
- Output delta changes
- E.g. binary search for sorted insert

Traditional: Re-run entire query on every change

→ O(n log n) for sorting, O(n × m) for joins

IVM: Incrementally update based on the change

→ O(log n) for sorted insert, O(max(n, m)) for incremental join

Submillisecond query execution even on large datasets

Final Problem: Initial Load Times

```
const issuesCollection = createCollection(
  queryCollectionOptions({
    queryKey: ['issues'],
    queryFn: api.getIssues,
    // ...
  })
)
```

```
const commentsCollection = createCollection(
  queryCollectionOptions({
    queryKey: ['comments'],
    queryFn: api.getComments,
    // ...
  })
)
```

- User just wants to skim a few issues...
- But they wait for ALL data to load first
- Issues & comments can be very large — doesn't scale!
  - Should load them lazily when needed

Solution: Sync Modes

Eager

All data loads upfront

Best for small datasets

On-Demand

Data loads when needed by a query

Best for large datasets

Progressive

Start on-demand, sync full data in background

Best for medium datasets

On-demand = Query-Driven Sync

The query expresses what data is needed → system fetches it just-in-time

Query-Driven Sync – Paginated Issues

```
// Paginated issues list with on-demand loading
const { data: issues } = useLiveQuery(
  (q) => q.from({ i: issuesCollection })
    .where(({ i }) => eq(i.projectId, projectId))
    .offset(pageIndex * pageSize)
    .limit(pageSize),
  [projectId, pageIndex, pageSize])

// Collection configured for on-demand sync
const issuesCollection = createCollection(
  queryCollectionOptions({
    syncMode: 'on-demand', // ← Key setting!
    // ...
  }))
```

How it works:

1. Collection starts empty
2. Query expresses demand for page 1
3. System fetches just that page
4. User scrolls → query demands page 2
5. System fetches next page

Query-Driven Sync – Paginated Issues

```
// Paginated issues list with on-demand loading
const { data: issues } = useLiveQuery(
  (q) => q.from({ i: issuesCollection })
    .where(({ i }) => eq(i.projectId, projectId))
    .offset(pageIndex * pageSize)
    .limit(pageSize),
  [projectId, pageIndex, pageSize])

// Collection configured for on-demand sync
const issuesCollection = createCollection(
  queryCollectionOptions({
    syncMode: 'on-demand', // ← Key setting!
    // ...
  }))
```

How it works:

1. Collection starts empty
2. Query expresses demand for page 1
3. System fetches just that page
4. User scrolls → query demands page 2
5. System fetches next page

Smart preloading → infinite scroll experience without loading spinners

Backend Integration Interface

```
// Backend integrations implement loadSubset
type LoadSubsetOptions = {
  where?: Expression<boolean>
  orderBy?: OrderBy
  limit?: number
}

interface BackendIntegration {
  loadSubset?: (opts: LoadSubsetOptions) => true | Promise<void>
}
```

TanStack DB is backend-agnostic
→ Relies on loadSubset to load data from the backend

TanStack Query Integration

```
interface BackendIntegration {
  loadSubset?: (opts: LoadSubsetOptions) => true | Promise<void>
}

// Query Collection implementation (simplified)
const loadSubset = async (opts: LoadSubsetOptions) => {
  const queryFn = getQueryFn(collection)
  // loadSubset options are passed to the query function
  const state = await queryFn({ meta: { loadSubsetOptions: opts } })
  // Update the local collection based on the query results
  collection.process(state)
}
```

TanStack Query Integration

```
interface BackendIntegration {
  loadSubset?: (opts: LoadSubsetOptions) => true | Promise<void>
}

// Query Collection implementation (simplified)
const loadSubset = async (opts: LoadSubsetOptions) => {
  const queryFn = getQueryFn(collection)
  // loadSubset options are passed to the query function
  const state = await queryFn({ meta: { loadSubsetOptions: opts } })
  // Update the local collection based on the query results
  collection.process(state)
}
```

TanStack Query is API agnostic → Predicates are passed to the queryFn

Paginated API Requests

Paginated issues query

```
const { data: issues } = useLiveQuery(
  (q) => q.from({ i: issuesCollection })
    .where(({ i }) => eq(i.projectId, projectId))
    .offset(pageIndex * pageSize)
    .limit(pageSize),
  [projectId, pageIndex, pageSize])
```

Paginated API Requests

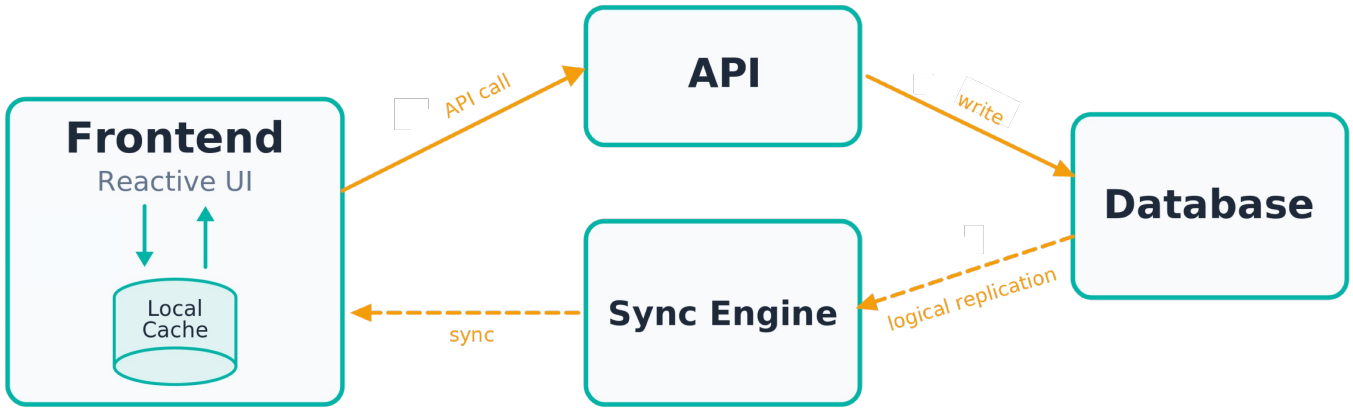
```
// Developer handles API translation in queryFn
const issuesCollection = createCollection(
  queryCollectionOptions({
    syncMode: 'on-demand',
    queryFn: async (ctx) => {
      const opts = ctx.meta?.loadSubsetOptions
      const projectId = getProjectId(opts.where)
      const page = computePageNumber(opts.limit)
      const res = await fetch(`/api/issues?project=${projectId}&page=${page}`)
      return res.json()
    }
  })
)
```

Paginated issues query

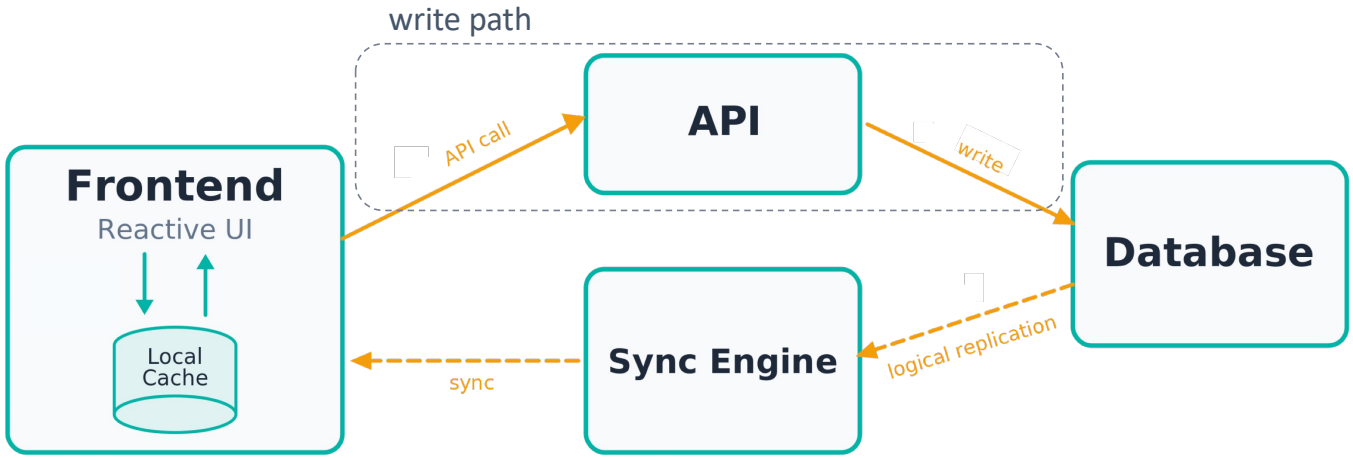
```
const { data: issues } = useLiveQuery(
  (q) => q.from({ i: issuesCollection })
    .where(({ i }) => eq(i.projectId, projectId))
    .offset(pageIndex * pageSize)
    .limit(pageSize),
  [projectId, pageIndex, pageSize])
```

Developer needs to handle predicates and translate into actual API calls

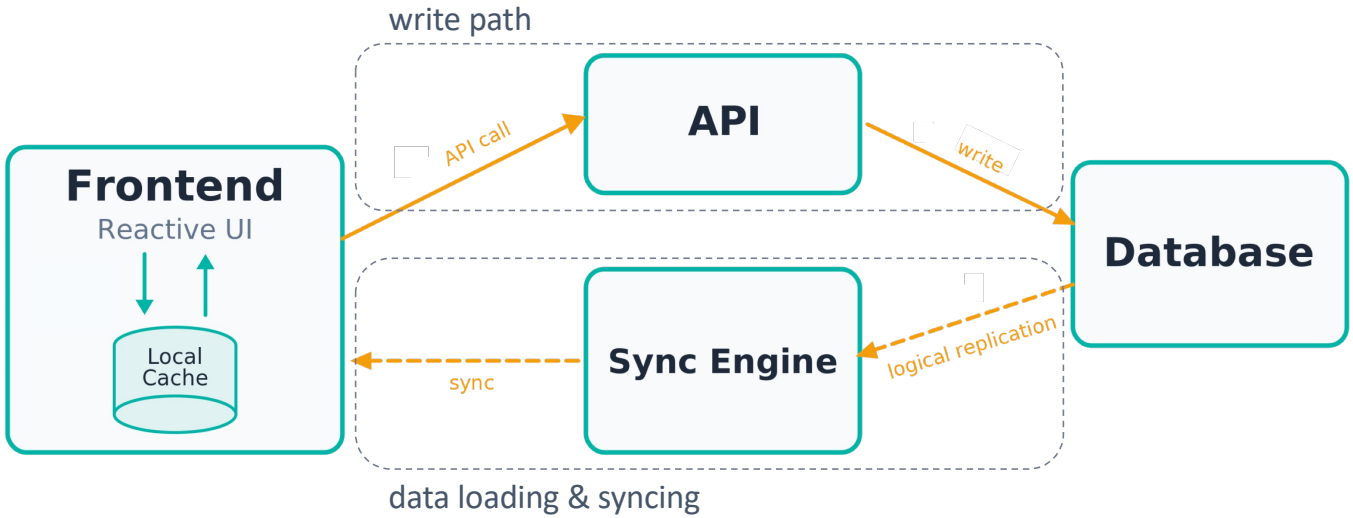
Swap Backend: Electric Sync



Swap Backend: Electric Sync



Swap Backend: Electric Sync



Swap Backend: Electric Sync

Before: Query Collections

```
queryCollectionOptions({
  syncMode: 'on-demand',
  queryFn: async (ctx) => {
    // Handle filters, ordering, pagination...
    const opts = ctx.meta?.loadSubsetOptions
    const page = computePageNumber(opts.limit)
    return fetch(`/api/issues?page=${page}`).json()
  }})
```

After: Electric Collections

```
electricCollectionOptions({
  syncMode: 'on-demand',
  shapeOptions: {
    url: '/api/electric',
    params: { table: 'issues' }
  },
  getKey: (i) => i.id,
})
```

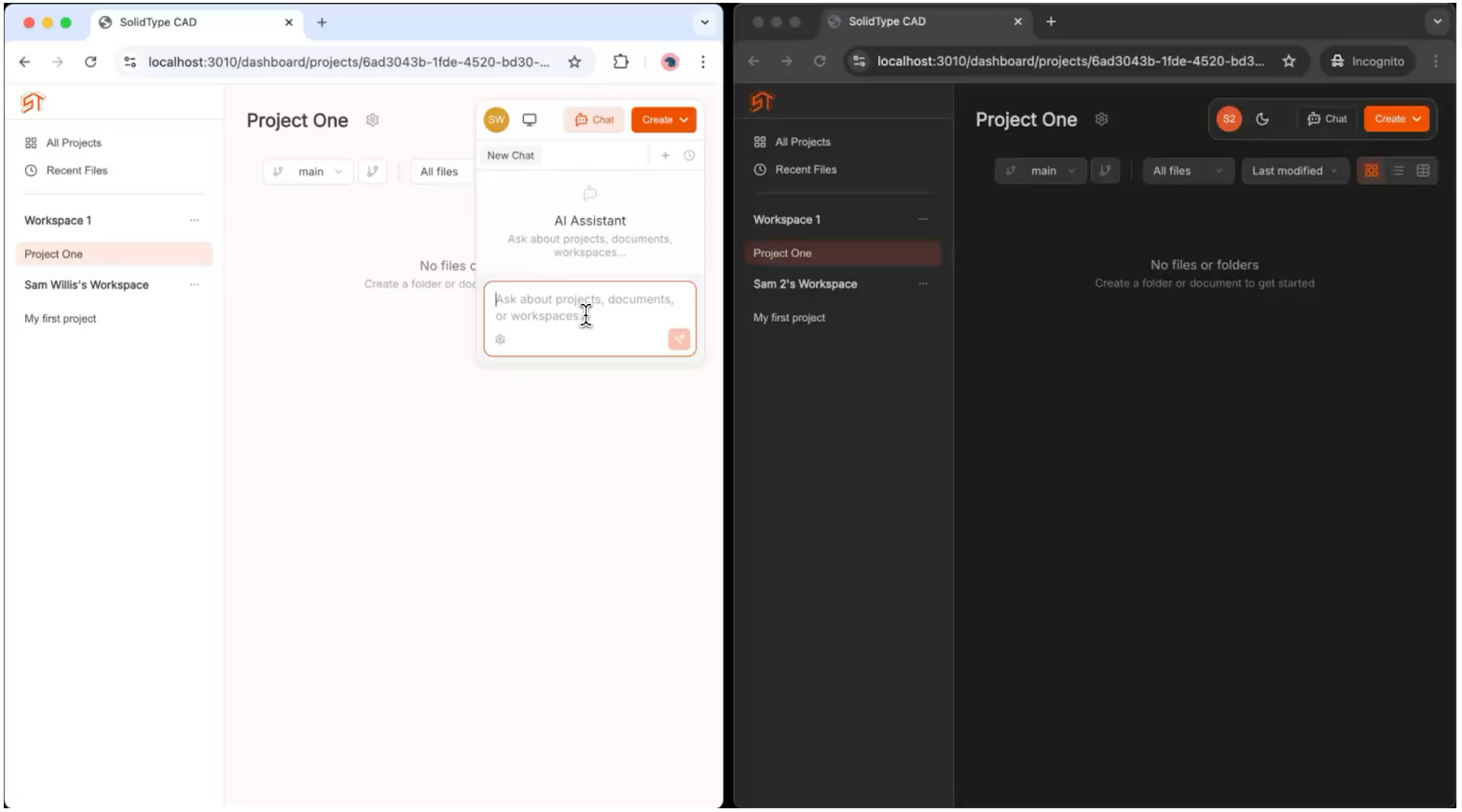
What changes:

✓ No more manual API translation — Electric is SQL-aware

✓ Real-time updates pushed from backend (no polling!)

✓ UI components & live queries stay exactly the same

Demo: SolidType CAD App



TanStack DB enables local-first apps with great UX and DX

Great UX:

- Query-driven sync with smart preloading → data ready when users need it
- IVM enables efficient queries on large datasets → fast, snappy apps

Great DX:

- Declarative queries — no networking, caching, syncing boilerplate
- Modular backend integrations that work with your existing stack

Thank you! • @KevinDP55 • electric-sql.com