Use Databases Without Putting Domain Logic in Them

February 09, 2023 • 6 minute read

Even though I'm very passionate about front-end development now, I started my career with different intentions. For the first few years, I worked predominantly on the back-end, and to this day, I move to the full-stack gray area when the situation demands it.

But databases are one part of the stack that's always given me trouble.

Code is ephemeral by nature. Every deployment can completely change its design or structure with little regard to its previous state. As long as it produces the desired output and side effects, you can refactor the implementation to support new requirements.

Data is not like this.

Data can't be (easily) changed to fit new business needs. You can't delete it and start from scratch because your users depend on it. And if a bug corrupts it, it's not just a matter of resolving the cause. You have to put the data in order too.

I don't have any claims of mastery over databases. I have a passable knowledge of SQL and a limited understanding of data modeling in NoSQL databases.

Still, to make my life easier, I've reached an important conclusion - utilize databases as much as possible without putting domain logic in them.

Logic in the Database

Most applications will want to execute logic as a result of something happening with their data. You can update a table holding statistics as a new record is saved or modify related records in a denormalized store when one is updated.

It's very appealing to implement this functionality in the database using some sort of trigger or stored procedure. After all, the store is doing all the work, and we would save an additional call from the application.

But this approach introduces maintainability problems and breaks the separation of concerns we hold so dear.

We're used to the application being the engine behind a product. It handles input from users, delivers output, and creates all the necessary side effects like database calls. If I have to work on a Node-based REST API, I would look through its handlers and files to see how it works

So it can be confusing if some of this functionality is implemented in the database instead.

Maintaining software is hard enough, so if we split our domain logic between the application and its storage, we're not doing ourselves a favor.

The functionality implemented outside our codebase is not subject to the same design rules and patterns we follow. Also, engineers proficient in the programming language we chose may lack the knowledge and understanding of the databases' mechanisms.

In effect, this is like introducing another language to your tech stack.

This leaves more room for error and makes our logic harder to test since it can't be easily mocked or executed with the rest of our application-level tests.

Utilize Without Adding Domain Logic

With all that said, we do want to take advantage of all our database's capabilities. If there's a way to do more with a single query, without writing logic in the store, then we should do so.

I recently worked on functionality that allowed users to like and unlike posts. However, I wasn't supposed to remove the entry in the database when they unliked something because, who knows, an analyst might find a correlation that leads to better profits based on that.

I needed a simple flag that specified the status of the entry, and there's a straightforward way to implement the liking functionality with it.

```
async function likePost(userId, postId) {
  // Check if the user has already interacted this post.
  const like = await repository.getLike(userId, postId)

  if (like.status === Status.Liked) {
    // Do nothing
    return
  }
  if (like.status === Status.Unliked) {
    // An entry exists but was unliked, so update its status
    return repository.updateLike(userId, postId, 'liked')
  }

  // Create a new entry
  return repository.createLike(userId, postId)
}
```

This leads to a more imperative implementation in which we have to make two queries in most cases and implement more functionality ourselves. It's closer to our natural way of thinking - if I were to describe the solution to this problem verbally, this is what I would probably say.

But an alternative would be to make our database do more work for us and add a unique index based on the `userId` and `postId`.

If we have a unique index on the post and user IDs we can always try to create a new entity without a check before that. If an entity already exists, we can write

an `ON CONFLICT` clause in the query and update it. This way, we utilize our database and remove complexity from our application at the same time.

```
async function likePost(userId, postId) {
  // Create an entry or mark it as liked if it already exists
  return repository.markAsLiked(userId, postId)
}

// And we need to add this to our SQL query once we have the unique index
;`
  ...
  ON CONFLICT (user_id, post_id)
  DO UPDATE saved = TRUE,
  ...
`
```

You could argue that having two separate handlers and methods would be a better design choice, but I'm just trying to illustrate the point.

We can also look for ways to improve how we fetch data. I write code the way I think about the logic in my head. But instead of firing multiple queries, we explore nested queries to reduce the calls to the database.

And it's not only SQL databases that we should aim to take full advantage of. DynamoDB, for example, has a `FilterExpression` parameter that allows you to filter results based on criteria that are not present in its primary index.

```
export const findAll = async () => {
  const result = await DynamoDB.query({
    TableName: 'items',
    FilterExpression: '#status = :status and #deadline > :deadline',
    ExpressionAttributeNames: {
      '#status': 'status',
      '#deadline': 'deadline',
    },
    ExpressionAttributeValues: {
      ':status': 'active',
      ':deadline': new Date().getTime(),
    },
  }).promise()

  return result.Items
}
```

While this gives you more flexibility, the tricky detail is that filtering happens only after your data is fetched. This is not different from fetching the results and filtering them inside your application. It just saves you some keystrokes.

```
export const findAll = async () => {
  const result = await DynamoDB.query({
    TableName: 'items',
    IndexName: 'status-deadline-index',
    KeyConditions: {
      status: {
        ComparisonOperator: 'EQ',
        AttributeValueList: ['active'],
      },
      deadline: {
        ComparisonOperator: 'GT',
        AttributeValueList: [new Date().getTime()],
      },
    },
  }).promise()

  return result.Items
}
```

Instead, we should focus on the table's design and leverage our indexes better for our access patterns. We can create secondary indexes that allow us to retrieve data based on different columns.

The Line Between Domain and Utility

It's not always clear where the line stands. In the `ON CONFLICT` example above, you could make a point that leaving this logic in the database is breaking the very rule we've been discussing so far. But to me, it all comes down to whether the database is making decisions of when and how something should be stored.

It should only act as a holder of information.

So in the example with `ON CONFLICT`, the database gives us a mechanism to describe what happens if a record that matches a certain constraint exists.

We're not letting any code live in it, and we're taking full advantage of its capabilities.

Reacting to Events

Our main reason to rely on the database to work for us remains - responding to something happening to the data. To retain the maintainability of our product, I would implement it entirely in the application, not relying on database triggers.

Triggers can hide important information about how our domain operates.

Eventually-consistent stores like DynamoDB remain a problem, though. We can't rely on a successful response from them to run functionality that expects the data to be stored because it still may not be present in all partitions.

In most cases, the data will be replicated very fast, but it remains an actual race condition.

In cases like this, we should try to reduce the exposure of business logic to the store. In the case of Dynamo, we'll have to use a trigger to make sure that the change has been propagated, but we can implement the functionality inside a lambda function.

The Bottom Line

I'm sure there are valid cases where moving functionality inside the database is the only reasonable way to solve a business problem. But in nine out of ten cases, you'd be better off not doing it.

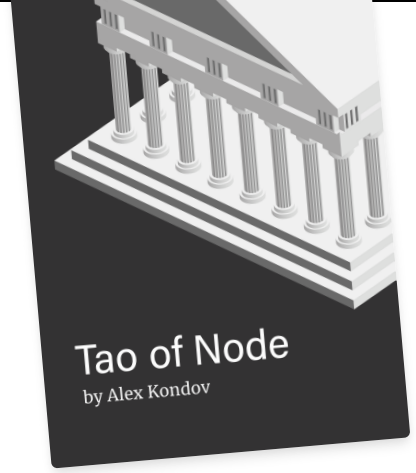
Utilize the full extent of your database's capabilities, but don't put domain logic in it.

Get Better at Software Design & Architecture

I send a twice-a-month newsletter about software design and architecture with a focus on JavaScript. It contains my latest article and occasionally some useful resources. No spam. Unsubscribe any time.

Your Email Address Subscribe

Not sure? View previous newsletters.



Tao of Node

Learn how to build better Node.js applications. A collection of best practices about architecture, tooling, performance and testing.