✨ Member-only story

Concurrency with MongoDB Explained

Secure Your Node.js Web Application — by Karl Düüna (29 / 78)

The Pragmatic Programmers · Follow
Published in The Pragmatic Programmers
6 min read · Jan 29, 2021

👈 *Ways to Mitigate Concurrency* | *TOC* | *Concurrency with MySQL Explained* 👉

Let's move back to practical usage and look at MongoDB since it's the most used NoSQL database for Node.js applications.

MongoDB doesn't guarantee ACID compliance[40] in all cases. For example, MongoDB doesn't have transactions and atomic functions that cover multiple documents. So if you have a database model where you need to avoid concurrency issues with multiple documents at the same time, you're in trouble. For example, you might have an e-commerce application that needs to decrement the number of products in the warehouse inventory while at the same time withdrawing money from the user's wallet.

ACID Compliance

When talking about concurrency and databases, the first thing that's usually looked at is whether the database is ACID compliant. That doesn't mean it must be able to withstand an acid attack; rather ACID stands for "Atomicity, Consistency, Isolation, Durability."

What these mean:

- Atomicity: operations must be atomic.
- Consistency: every transaction must leave the database in a consistent (valid) state.
- Isolation: transactions cannot interfere with each other.
- Durability: effects of transactions must persist through crashes.

The reasoning is that concurrency issues are far easier to deal with in an ACID-compliant database.

Faced with this situation you are left the option to modify the database schema to collect information into single documents so that you can use atomic functions, or you can create a locking mechanism for a collection of documents. MongoDB has good information in its documentation about two-phase commit,[41] which is essentially a locking mechanism. Finally, you could also use another database, either in conjunction with MongoDB or completely migrate to that.

Mongo vs. Other

INFORMATION

I think this is a good spot to point out that just because you're using Node.js in your application, it doesn't mean that a NoSQL database is the best solution for you — database selection should be determined by your application and information types and connections.

In this section, we'll fix the example using locks and atomic functions. Locking is the more general solution because you can extend it over several operations and more complex applications; this specific example benefits more from atomic functions. So let's look at the simpler atomic approach and then see how you can use locking.

To use atomic functions we need to modify our logic to deduct the amount first and then perform the operation. If anything goes wrong, we reimburse the amount:

chp-6-concurrency/concurrency-wallet-mongo-delay-atomic.js

```
 // Our processing function
 function processCall(cb) {
     // Add delay of 5 seconds here - imitating processing of the
request
     setTimeout(cb, 5000);
 }

 app.post('/:name', function(req, res, next){
     var amount = Math.abs(req.body.amount);

»    // Search by name and amount greater than or equal to requested
»    var search = {name: req.params.name, amount: {$gte: amount}};
»    // Increment by negative amount
»    var update = {$inc: {amount: -amount}};
»    Wallet.findOneAndUpdate(search, update, function (err, wallet) {
         if (err) { // Something went wrong with the query
             next(err);
             return;
         }
         if(!wallet) {
             res.send(400, 'Insufficient funds or not found');
             return;
         }

         processCall(function (err) {
»            if(err) {
»                // Process failed so reimburse
»                wallet.amount += amount;
»                wallet.save(function (rErr, updatedW, rowsAffected) {
»                    if(rErr || rowsAffected !== 1) {
»                        //TODO: This needs careful handling
»                        console.error('Reimbursement failed');
»                    }
»                    res.send(500);
»                });
```

```
»                 return;
»             }
              res.redirect('/' + req.params.name);
          });
      });
  });
```

And with these simple modifications our application is secure against concurrency attacks. All the calculations are atomic and the results will be correct.

Now let's expand our database model to include a lock model so that you can see how to achieve the same thing with locking:

chp-6-concurrency/concurrency-wallet-mongo-delay-lock.js

```
var mongoose = require('mongoose');
var args = require('minimist')(process.argv);

if(!args.d) {
    console.log('This example requires the -d (mongoose db) command
line variable');
    process.exit();
}

// Connect to mongoose db
mongoose.connect(args.d);
mongoose.connection.on('error', function (err) {
    console.error('connection error:' + err);
    process.exit();
});

// Define wallet model
var walletSchema = new mongoose.Schema({
    name:  { type: String, required: true, index: { unique: true } },
    amount: { type: Number, required: true}
});
var Wallet = mongoose.model('Wallet', walletSchema);

// Define wallet model
var lockSchema = new mongoose.Schema({
    name:  { type: String, required: true, index: { unique: true } }
});
var Lock = mongoose.model('Lock', lockSchema);

Wallet.remove().exec(); // Delete all previous Wallets.
```

At this point, we'll create and apply the mechanism to lock the resources and release them afterward:

chp-6-concurrency/concurrency-wallet-mongo-delay-lock.js

```
 // Our processing function
 function processCall(cb) {
     // Add delay of 5 seconds here - imitating processing of the
request
     setTimeout(cb, 5000);
 }

»function aquireLock(name, cb) {
»    // We will use the fact that name is unique and
»    // so insert will fail if a lock exists
»    Lock.create({ name: name }, function (err, lock) {
»        if(err) {
»            cb(new Error('Failed to aquire lock'));
»            return;
»        }
»        cb(null, lock);
»    });
»}
»
»function releaseLock(name, cb) {
»    Lock.findOneAndRemove({name: name}, cb);
»}

 app.post('/:name', function(req, res, next){
     var amount = Math.abs(req.body.amount);

»    aquireLock(req.params.name, function (err) {
»        if(err) {
»            res.send(409, 'Already processing');
»            return;
»        }
         Wallet.findOne({name: req.params.name}, function (err,
wallet) {
             if (err) { // Something went wrong with the query
                 next(err);
                 return;
             }
             if(!wallet) {
                 res.send(404, 'Not found');
                 return;
             }
             if(wallet.amount < amount) {
                 res.send(400, 'Insufficient funds');
```

```
                    return;
                }
                processCall(function () {
                    wallet.amount -= amount;
                    wallet.save(function (rErr, updatedW, rowsAffected) {
                        if(rErr || rowsAffected !== 1) {
                            res.send(500, 'Withdrawal failed');
                            return;
                        }
»                       releaseLock(req.params.name, function (err) {
»                           if(err) {
»                               //FIXME: We should definitely handle the
error here
»                               console.error(err);
»                           }
»                           res.redirect('/' + req.params.name);
»                       });
                    });
                });
            });
        });
    });
```

We can acquire a lock atomically to the wallet and perform operations before releasing the lock. Because this example doesn't take into account any potential errors, we can wind up with eternally locked resources. We'll solve that problem by adding a timeout to the lock:

chp-6-concurrency/concurrency-wallet-mongo-delay-lock-improved.js

```
// Define wallet model
var lockSchema = new mongoose.Schema({
    name:  { type: String, required: true, index: { unique: true } },
    timestamp: {type: Number}
});
var Lock = mongoose.model('Lock', lockSchema);
```

Now when acquiring the lock, we'll have to adjust for the timestamp to prevent eternal locks:

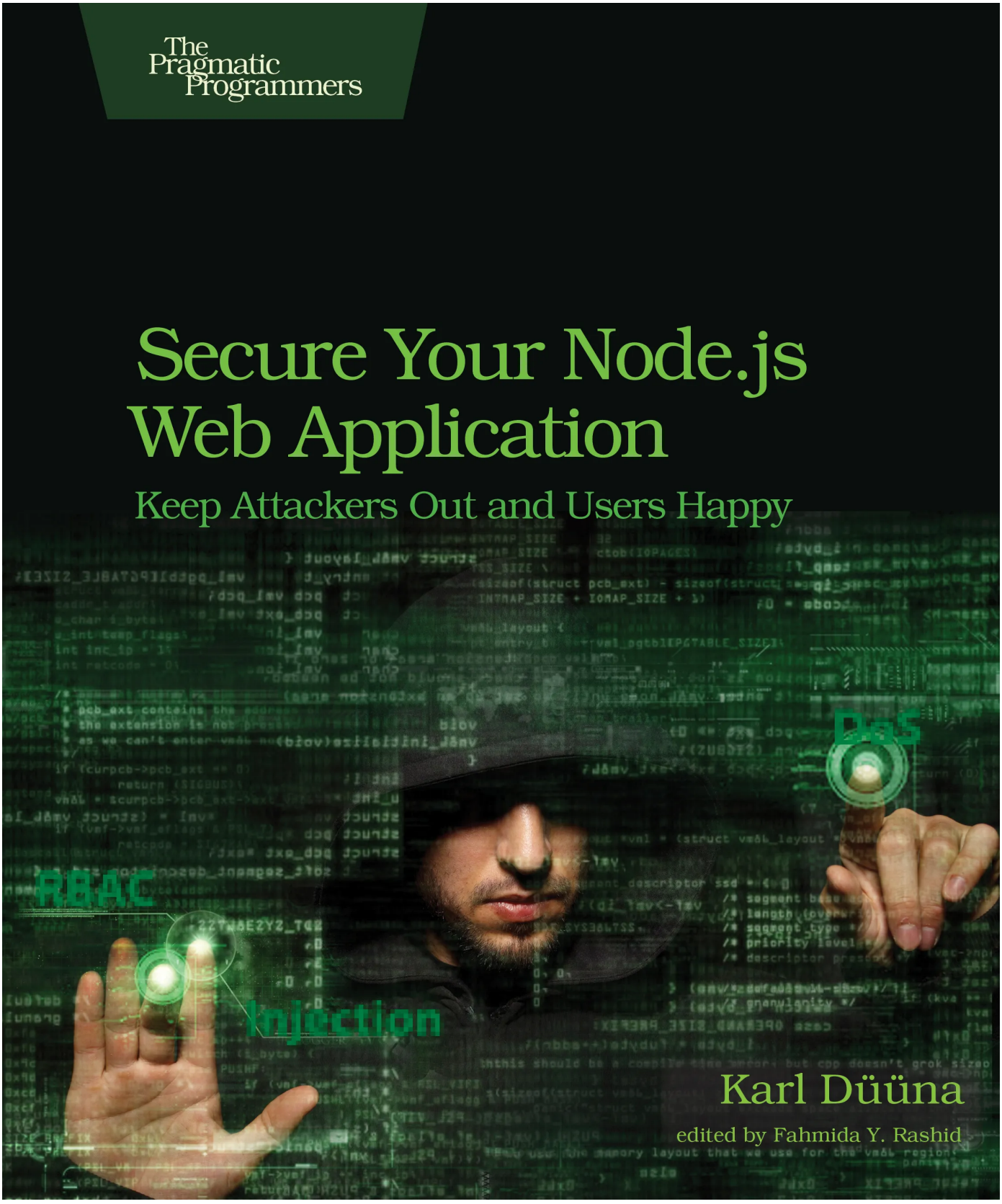
chp-6-concurrency/concurrency-wallet-mongo-delay-lock-improved.js

```
 // Our processing function
 function processCall(cb) {
     // Add delay of 5 seconds here - imitating processing of the
request
     setTimeout(cb, 5000);
 }

 function aquireLock(name, cb) {
»    var now = Date.now();
»    var expired = now - 60 * 1000;
»
»    // The basics of this command is that we either:
»    // 1. Find an old lock and update it with a new timestamp
»    // 2. Don't find one, in which case we try to insert
»    //    This will either:
»    //    2.1 fail, because of unique index - a lock exists
»    //    2.2 succeeds - a new lock is created
»    Lock.findOneAndUpdate({
»        name: name,
»        timestamp: {$lt: expired} //Include locks that are too old
»    }, {
»        timestamp: now
»    }, {
»        'new': true,   // return new doc if one is upserted
»        upsert: true // insert the document if it does not exist
»    }, function (err, lock) {
         if(err) {
             cb(new Error('Failed to aquire lock'));
             return;
         }
         cb(null, lock);
     });
 }


 function releaseLock(name, cb) {
     Lock.findOneAndRemove({name: name}, cb);
 }
```

Both of these solutions are valid. Atomic functions require you to change your operational logic only slightly, but you can see that locking mechanisms are more versatile.

👈 *Ways to Mitigate Concurrency* | *TOC* | *Concurrency with MySQL Explained* 👉

Secure Your Node.js Web Application by Karl Düüna can be purchased in other book formats directly from The Pragmatic Programmers. If you notice a code error or formatting mistake, please let us know here so that we can fix it.



Kdnodesec

Follow

Written by The Pragmatic Programmers

9.5K Followers · Editor for The Pragmatic Programmers

We create timely, practical books and learning resources on classic and cutting-edge topics to help you practice your craft and accelerate your career.

More from The Pragmatic Programmers and The Pragmatic Programmers

The Pragmatic Programmers in The Pragmatic Programmers

Fight Unplanned Work, the Silent Killer of Projects

An excerpt from Your Code as a Crime Scene, Second Edition by Adam Tornhill

5 min read · 1 day ago

84 2

Margaret Eldridge in The Pragmatic Programmers

Programming Books for Summer 2023

From The Pragmatic Bookshelf

3 min read · Jun 23

83

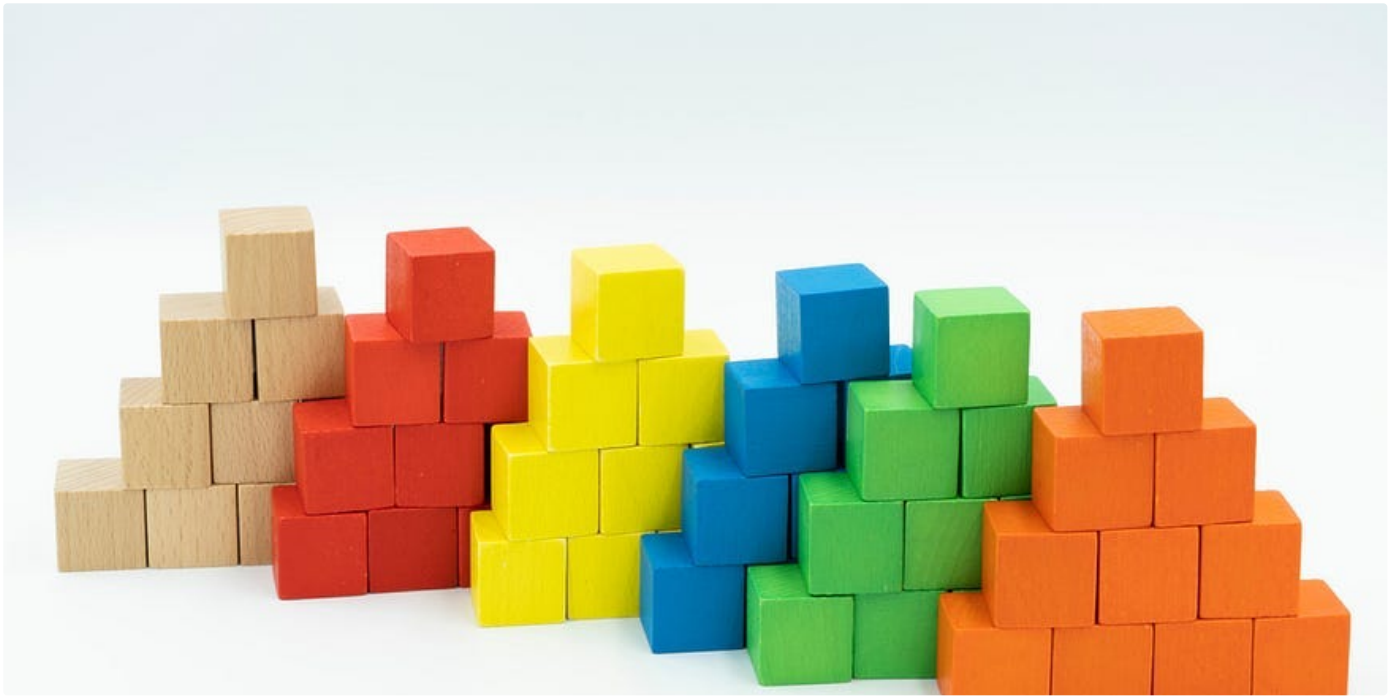
Mike Riley in The Pragmatic Programmers

Five Promising Technologies to Watch

Noteworthy Tech That Inspires

4 min read · Jun 6

102 1



The Pragmatic Programmers in The Pragmatic Programmers

Compiling Your Go Application for Containers

An excerpt from Powerful Command-Line Applications in Go by Ricardo Gerardi

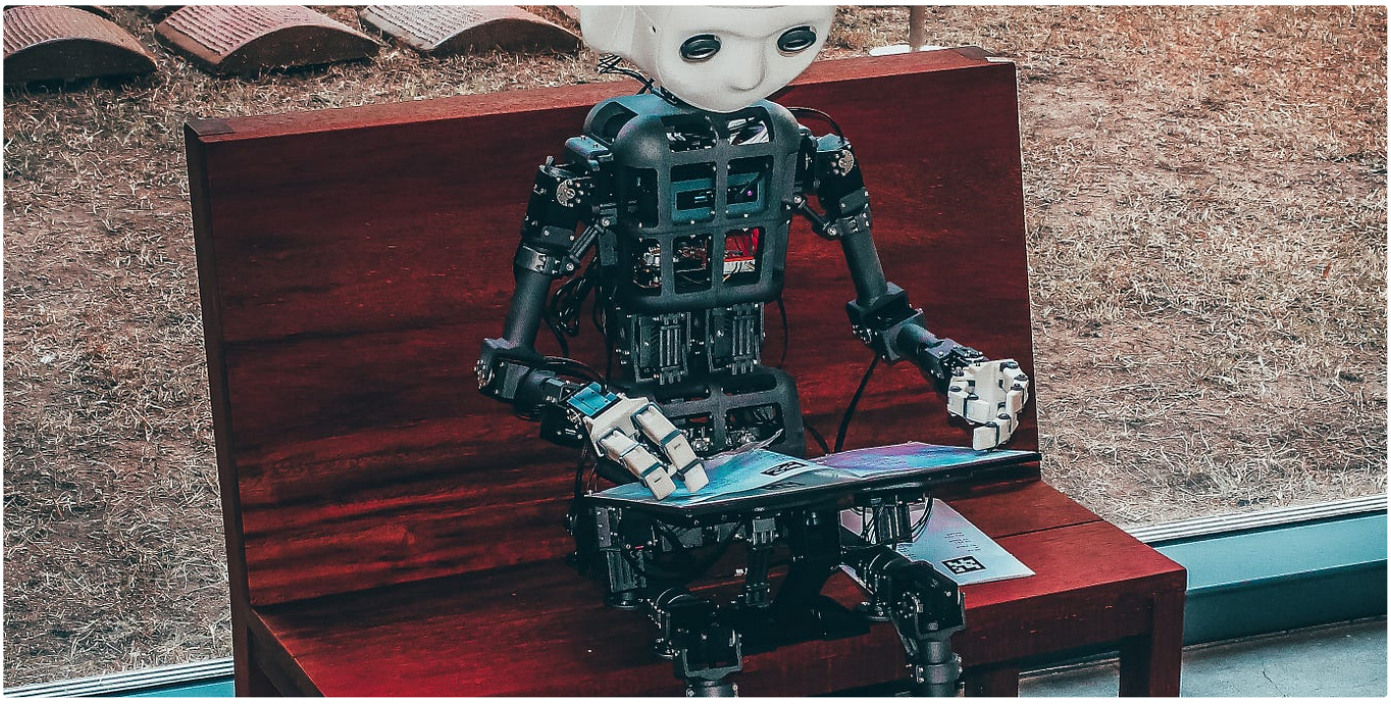
8 min read · Jun 15

59

See all from The Pragmatic Programmers

See all from The Pragmatic Programmers

Recommended from Medium



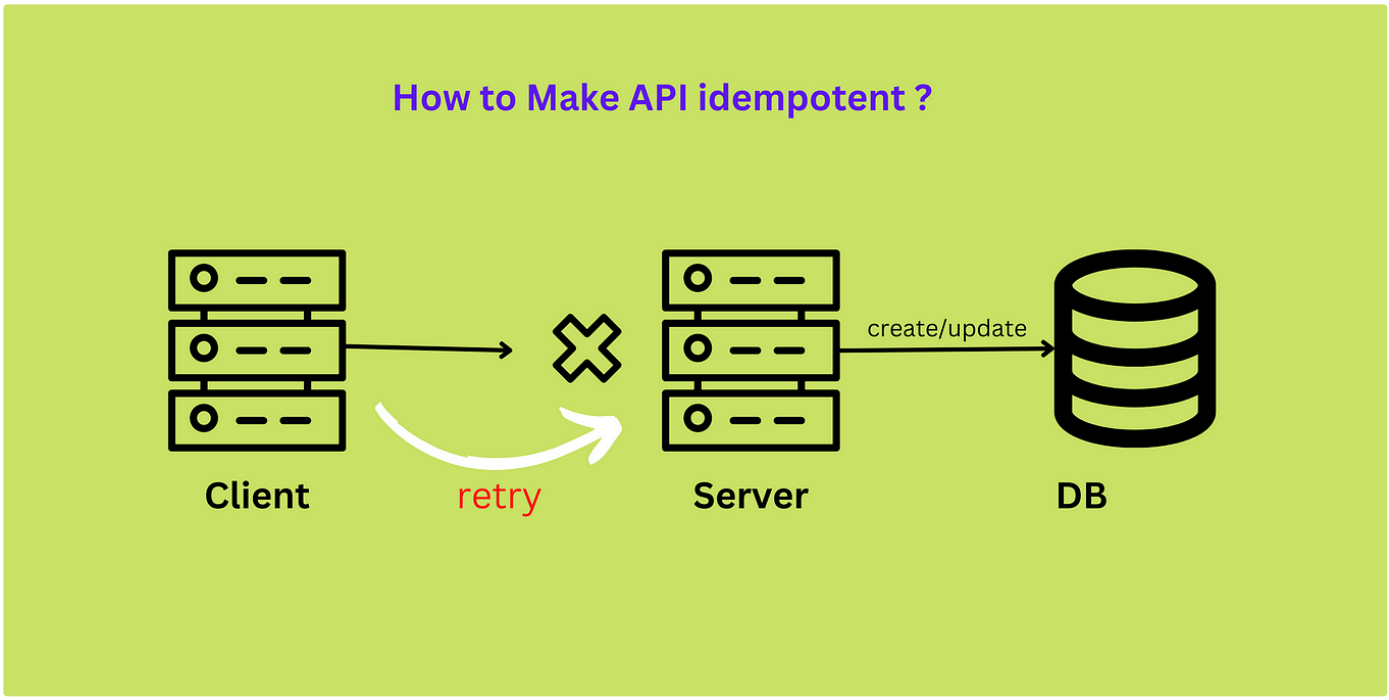
Aparna Jain

Is AI(chatGPT) Better Than An Experienced Engineer For Career Advice?

The chat bot blew my mind, but is it smarter than my life experience?

· 6 min read · Jan 2

19 2



Suraj Mishra in Level Up Coding

How to Implement Idempotent API (Part 1)

Discussing idempotency concept and implementation design

· 5 min read · Jan 27

100 1

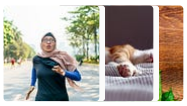
Lists

Staff Picks

372 stories · 134 saves

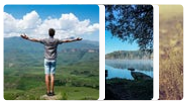
Stories to Help You Level-Up at Work

19 stories · 125 saves



Self-Improvement 101

20 stories · 214 saves



Productivity 101

20 stories · 231 saves

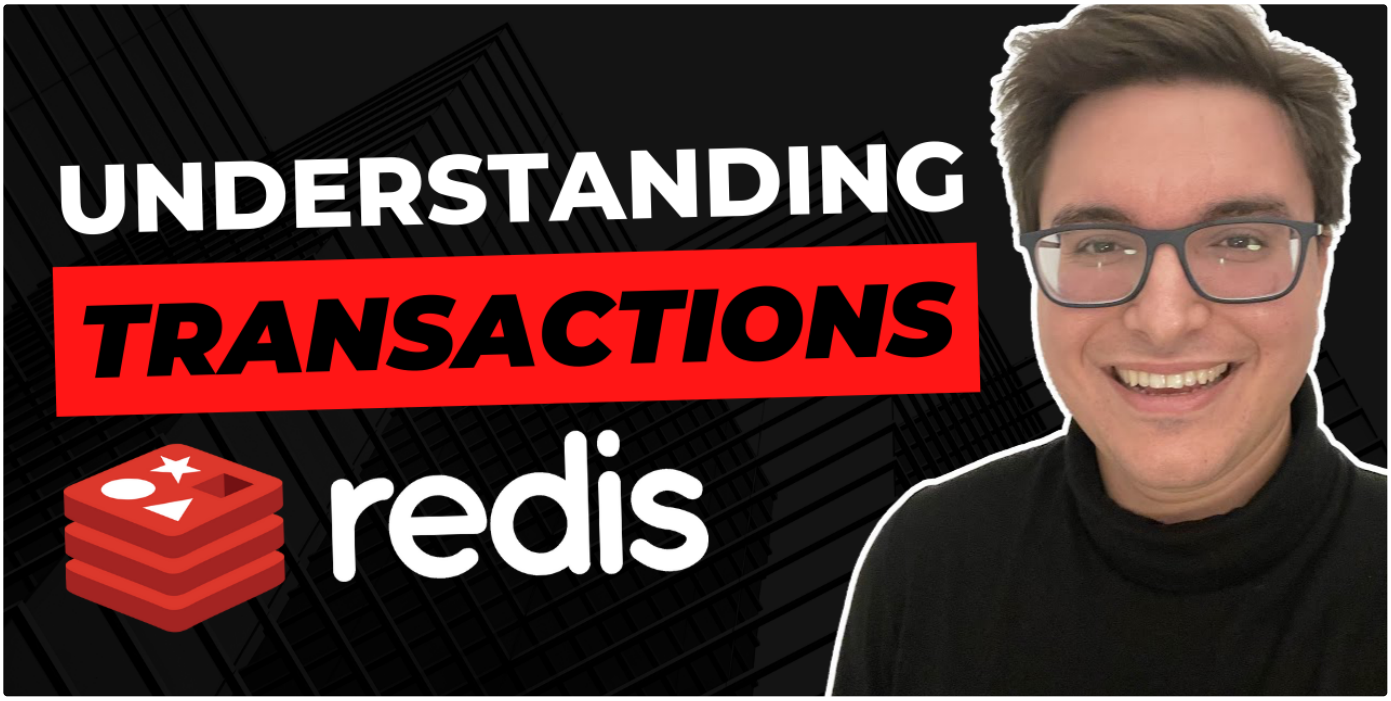
JP Brown

What Really Happens to a Human Body at Titanic Depths

A Millisecond-by-Millisecond Explanation

· 4 min read · Jun 23

26K 319

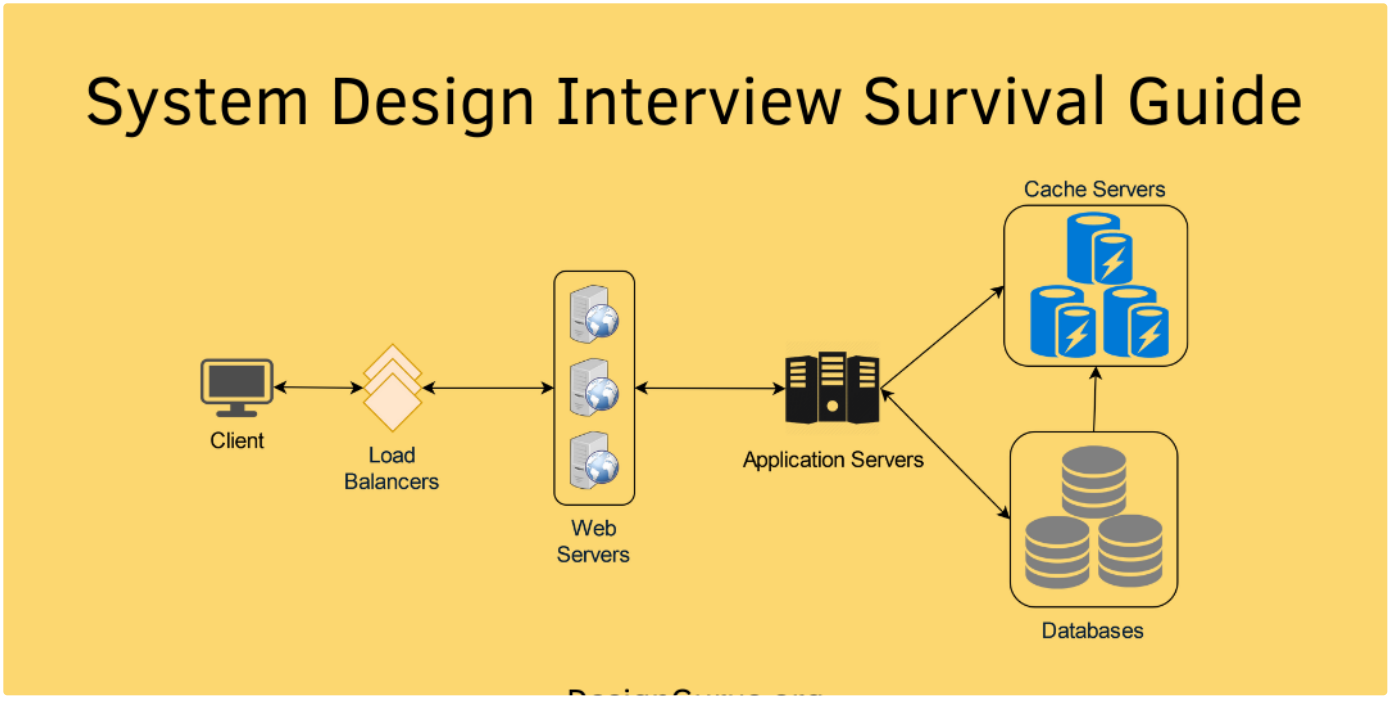


Raphael De Lio

Understanding Transactions in Redis (How to

This content is also available as a YouTube video. You can watch it here.

· 6 min read · Jan 16



Arslan Ahmad in Level Up Coding

System Design Interview Survival Guide (2023): Preparation Strategies and Practical Tips

System Design Interview Preparation: Mastering the Art of System Design.

✨ · 14 min read · Jan 19

1.8K 8

Israel Josué Parra Rosales in Dev Genius

Tutorial - Elastic Compute Cloud (EC2)

Creating EC2 instances from zero

✨ · 8 min read · Feb 25

8

See more recommendations