PROJECT

RICOCHET

HOW TO ADDRESS THE

TOP 10 METEOR PERFORMANCE PROBLEMS

HOW TO ADDRESS THE TOP 10 METEOR PERFORMANCE PROBLEMS

When Meteor was first released in 2012, it represented a fairly ground-breaking approach to building web and mobile apps. All of a sudden, it became much more simple to create real-time applications using a single language on both the client side and server side.

Many, like us at Project Ricochet, experimented with Meteor and were impressed by how things just "magically" worked. Even in those early beta releases! Key features included database triggered live-page updates, data synchronization, live queries, latency compensation, hot code pushes, and more. Before long, Meteor developers (ours included) started turning their ideas into actual production apps.

Perhaps the true magic of Meteor is how it helps simplify a set of complex technologies behind the framework. This gives developers capabilities that previously were not possible – or required a large amount of resources.

Certainly, that newfound simplicity made it easier for developers to create amazing real-time applications. However, if you weren't careful, application performance could slip in fairly rapid fashion. Add a few concurrent users — and you could suddenly encounter all sorts of issues.

In four short years Meteor has greatly matured to become a polished end-to-end stack. The community and knowledge base around Meteor has also grown immensely.

Here at Project Ricochet, our Meteor knowledge has dramatically improved by tackling our own challenges and learning from others. We've now written over 500,000 lines of Meteor code. Early on, we recognized the potential value of

learning more about the framework. From the day we started experimenting with Meteor, we have also learned where and how to minimize performance problems.

Set aside Meteor for a moment. *Any* web or mobile application that takes on a growing number of users can suffer performance issues. More users can readily cause bottlenecks to key resources like the CPU, network, memory, or the database. And, with more demand on resources, users begin to experience longer response times. Client and server machines can start maxing out on CPU and memory usage.

Well, Meteor applications are no different.

Here, we'll present the top 10 major Meteor performance issues our team has encountered over the past few years. We'll also offer solid ideas on how to minimize them or avoid them altogether. These issues occur in different points of the stack – from the database all the way to the devices the apps run on.

Certainly, these aren't the *only* performance pitfalls possible with Meteor. But we believe these are some of the most critical issues to watch out for when developing your Meteor application. We hope this list can help you more easily navigate through the big challenges we've had to face.

At first, the Meteor feature that perhaps impresses most is the fact that applications are real-time *by default*. Whether back in 2012 or even today, seeing your database updates display in the browser automatically is both useful and impressive. So now, when aspects of an app you want to create need to function in real-time, the Meteor framework is a natural choice for development.

Behind the scenes, delivering this real-time capability is no easy undertaking. It places a high demand on the CPU, memory, network, and the database to work. This is why your snappy real-time app can rapidly decelerate as more users start coming on board.

So, how can you get the best performance from your Meteor app as you develop? You can start by increasing your awareness of the following issues:

ISSUE #1: PREVENT METEOR FROM PUBLISHING TOO MUCH DATA

You may have heard this one already. It's one of the first performance optimizations a Meteor developer tends to discover. But given the performance cost from tracking a large amount of real-time data, it's definitely worth a reminder.

With Meteor, the server maintains a copy of every client's version of the database. So, the more clients connected, the more copies it has to track. Now, having many clients means your app is being used and appreciated. So, you probably don't want to reduce the number of clients. But you should reduce the potential performance hit by minimizing the amount of data needed by each client.

We can do this via our Publish functions. The Publish functions make real-time data available for each client to use. So, if you can implement the Publish function to send only what is absolutely needed, the amount of data is reduced.

To do this, you must first understand why every part of the data set is needed and how they will be used. Is the data associated with a certain user? If so, is only that user's data needed? Will all the fields be required for display or processing on the client? Or, just a certain subset? Will all records be displayed at the same time? Or, will the user "page through" the data set? Asking questions like these will provide ideas on how to start narrowing your publication.

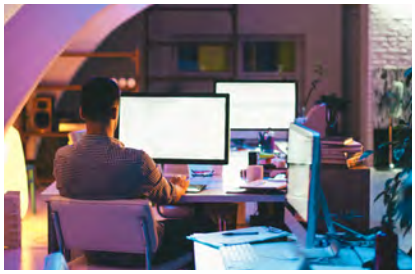
Let's say you're developing a chat app. If only a list of authors participating in a certain channel are needed, then you can use field filtering in your publish function to send back only their names:

```
Meteor.publish("channelParticipants", function(channelId) {
      return Messages.find({
            channelId: channelId
      }, {
            fields: {
                  authorName: 1
            }
      });
});
```

If your publication needs to support infinite-scroll type of pagination, you can use the popular percolate:paginated-subscription package. This makes it easy to request a limited number of documents at a time.

ISSUE #2: MANAGE COUNTING AND AGGREGATION FUNCTIONS ON THE CLIENT SIDE

Ever need information about the collection as a whole — such as the number of records? One inefficient way to do this is to publish a subset of the collection (or even worse, the entire collection) to the client for a count to be performed there. The problem with that approach is related to Issue #1. A potentially large amount of data will be published, taxing both the server and network.



In this case, it's better to run a query on the server side and perform a count there. This eliminates the need to send the entire collection to the client. You can implement this using a Meteor method:

```
Meteor.methods({
    'numberOfMessages' () {
          return Messages.find().count();
    }
});
```

If you absolutely need a real-time count, you can use the tmeasday:publish-counts package which does the same thing via a publication. Just be aware that it's not designed to return a count for large datasets. From its documentation:

> Publish-counts is designed for counting a small number of documents around an order of 100. Due to the real-time capability, this package should not be used to count all documents in large datasets. Maybe some, but not all. Otherwise you will maximize your server's CPU usage as each client connects.

The same goes for other aggregation operations. Depending on the circumstance, attempting to perform aggregation-related functions like sums or even counts on the client often involves a large dataset. It's more efficient to perform aggregation

on the server and provide the client with the calculated values.

There are several packages which provide MongoDB aggregation support on Meteor's server side. One popular one is meteorhacks:aggregate.

In our example, if the client needs to display the number of messages per topic (instead of requesting all messages *then* calculating this value on the client) this aggregate package can be used to determine that on the server side:

```
Messages.aggregate(
    [
        {
          $group : {
             _id : "$channelId",
             channelCount: { $sum: 1 }
          }
        }
    ]
)
```

ISSUE #3: PUBLISH DATA IN REAL-TIME ONLY WHEN NEEDED

So, Meteor makes your app real-time by default. But as we've also learned, performance can be penalized if you're not careful. What if there is data we don't need to be real-time in the first place? How can we prevent the server from unnecessarily sending updates to the client since we're not going to use it?

Consider our chat app. We'd like the homepage to display the most recently created channels, but it's not required for this list to update if the user stays on the page indefinitely.

To improve this, you could create a Meteor method that performs the query on the server, then returns an array of channels. Method responses are not reactive data sources. However, we may lose the convenience of being able to further query the response using the Collection API. Instead, we can provide it as a publication. This enables the client to access it as a data collection, but prevents further changes on the dataset —

effectively disabling its reactiveness:

```
Meteor.publish("newestChannels", function() {
      const docs = Channels.find({}, {
            sort: {
                  createdAt: -1
            },
            fields: {
                  _id: 1,
                  name: 1
            },
            limit: 10
      }).fetch();
      docs.forEach(channel => this.added("channels",
channel._id, channel));
      this.ready();
});
```

In the above code, the initial query gets the names of the 10 most recently created channels. However, instead of returning a cursor to the dataset, it fetches all the records, and copies it to separate collection that the client can subscribe to. Because this collection will not be updated (notice there are no updated or removed callbacks implemented) this data set will not change.

ISSUE #4: PREVENT A HIGH NUMBER OF OBSERVERS

When a client subscribes to a publication, observers are set up on the server to track any data changes that might occur on the queries involved. The more connected clients there are, the more subscriptions are established, which increases the number of observers. These observers place more demands on the CPU and the network between Meteor and MongoDB.

Meteor attempts to optimize this situation by checking to see if there are any active matching observers. More specifically, if an existing observer was created from a query that uses the same collection, same selector, and same set of query options, then it can be reused. So, you can help improve Meteor's efficiency here by creating publications that increase the chance for observer reuse.

For example, try to normalize your query or options. Rather than asking it to query based on a client-provided limit like so:

```
Meteor.publish('recentMessages', function(limit) {
   return Messages.find({}, {limit: limit});
});
```

… round up the limit to the nearest 10:

```
Meteor.publish('recentMessages', function(limit) {
      var base = 10;
      var normalizedLimit = limit + (base - (limit % base))
      return Messages.find({}, {limit: normalizedLimit});
});
```

The client can then limit it further if necessary. Doing this means Meteor can reuse the same observer for two subscription requests if they passed in limits of 15 and 20, for example.

You can measure how much observer reuse occurs in your app with Kadira, a performance monitoring platform for Meteor. It's a must have for monitoring your production apps. Kadira includes an Observer Reuse metric that can be tracked over time. It helps you identify which publications are candidates to optimize.

ISSUE #5: OPTIMIZE HOW METEOR HANDLES SUBSCRIPTIONS

Meteor uses a pub/sub approach to handling application data. We covered publications in the last example. What about subscriptions?

Every subscription triggers the set up and flow of real-time data between the client and server. As more subscriptions become active, or as the data set of each subscription grows, more demands are placed on the entire system. So, applying optimizations to your subscriptions can really help with performance too.

In a way similar to what Meteor does for publications, Meteor applies optimizations for subscriptions too, when it can.

From the Meteor Docs:

> If you call Meteor.subscribe within a reactive computation, for example using Tracker.autorun, the subscription will automatically be cancelled when the computation is invalidated or stopped; it is not necessary to call stop on subscriptions made from inside autorun. However, if the next iteration of your run function subscribes to the same data set (same name and parameters), Meteor is smart enough to skip a wasteful unsubscribe/resubscribe.

But, we can also help Meteor here. If normalizing the parameters isn't feasible on the `recentMessages` publication, as showcased in the last example, we can apply the normalization in the subscription instead.

```
Meteor.subscribe("recentMessages", normalizedLimit);
```

It's not as optimal as normalizing on the publication, as it puts the responsibility on the client subscription. But it does offer the flexibility for the client to choose, based on the situation, whether it's acceptable to normalize or not.

ISSUE #6: DISABLE UNNECESSARY REACTIVITY

Sometimes there's too much of a good thing.

Anytime a publication's dataset changes, computations observing that dataset will rerun. This allows the client to react appropriately. But if the data changes frequently, the computation will have to rerun frequently as well. If there's a lot is happening in the computation, this can slow down the app.

Consider our chat app example again. Suppose we want to perform some operations on the content of the most recent 20 messages. We could set up the subscription and computation as follows:

```
Meteor.subscribe("recentMessages", 20);
Tracker.autorun(function() {
   var recentMessages = Messages.find({}).fetch();
   // some very expensive operations
})
```

Trouble is, this computation will rerun with any changes to the dataset, even if we don't need the values that were changed. So, if each message has a views field that gets incremented when a message is viewed, that update will trigger the computation to rerun. If the computation doesn't even use the views value, that's a waste.

So, why not filter your query to only the fields that are needed?

```
Tracker.autorun(function() {
   var recentMessages = Messages.find({}, {fields:
{content:1}}).fetch();
   // some very expensive operations
})
```

Now, *only if the content field is updated* will this computation rerun. If you don't need reactivity at all, then disable it by using Tracker.nonreactive(). This can come in handy, for example, in functions that are reactive computations by default, like template helpers.

```
Template.recentMessages.helpers({
 messages: function() {
   return Tracker.nonreactive(function() {
     return Messages.find({}, {fields: {content:1}}).fetch();
   });
 }
});
```

Finally, if your computation requires some data to be reactive and some not, then you can disable reactivity per query. In the following example, we use react-meteor-data's `createContainer` function to create a React container component, `RecentMessagesContainer`. For our purposes, we don't need this computation to rerun at all if the `recentMessages` dataset changes. So, we can disable it by passing in the `reactive: false` option.

```
export default RecentMessagesContainer =
   createContainer(({}) => {
     Meteor.subscribe('recentMessages', 20);
     const recentMessages =
         Messages.find({},
             {fields: {content:1}, reactive: false}).fetch();
     // some very expensive operations and other reactive
queries
     return {
       recentMessages
     };
   }, RecentMessages);
```

Remember, in Meteor, apps are real-time by default. To improve performance, analyze the needs of your app and its users, then selectively optimize reactivity using the strategies suggested in these last few sections.

ISSUE #7: REDUCE UNNECESSARY POLLING FOR DATABASE CHANGES

Meteor will automatically poll the database for changes on an ongoing basis. This isn't necessarily a bad default setting for when there's frequent updates. But, it's a waste of resources on both the Meteor server and MongoDB whenever data changes don't occur.

As far back as version 0.7, Meteor could read the MongoDB "operations log." It's a special collection that logs all the write operations occurring in your database. From this log, Meteor will notice instantly when an update occurs, and call the necessary callbacks to trigger reactive updates.

"Oplog tailing" is actually enabled by default on your local development environment, but in your other environments – like Production – you'll need to enable it manually. You can do so by setting the MONGO_OPLOG_URL environment variable:

```
MONGO_OPLOG_URL=mongodb://oplogger:PasswordForOplogger@
mongo-server-1.example.com,mongo-server-2.example.com,mongo-
server-3.example.com/local?authSource=admin&replicaSet=replic
aSetName
```

There are certain types of queries, however, where Meteor can't use the oplog to watch for updates. From Meteor's OplogObserveDriver documentation, these include:

- Selectors containing geospatial operators, the $where operator, or any operator not supported by Minimongo (such as $text)
- Queries specifying the skip option
- Queries specifying the limit option without a sort specifier or with a sort based on $natural order

- Field specifiers and sort specifiers with $ operators such as $slice or $natural
- Calls to observeChanges using "ordered" callbacks addedBefore and movedBefore. (The implicit call to observeChanges which occurs when you return a cursor from a publish function does not use the ordered callbacks.)

So, avoid these if you can!

Optimizing oplog usage helps facilitate quicker reactive updates and lower CPU usage on the server and MongoDB for your app.

ISSUE #8: IDENTIFY AND RESOLVE SLOW READS AND WRITES

READS

As your database grows, likely you'll notice certain queries take longer to execute. More data means more information for MongoDB to process your query against. These issues can often catch you off guard, because they hardly ever happen in a local development database featuring a smaller amount of test data. However, in Production this degradation can happen quickly. Especially with a significant and sudden increase in users.

The best way to speed up your queries is to add indexes to your collections. These are special data structures that store a small portion of the collection's data set in an easy to traverse form. All your queries should be supported by an index. Without indexes, MongoDB must scan every document in a collection to find the ones that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

For example, if you have the following query:

```
Messages.find({"authorId": authorId} );
```

This can drastically slow your app down as more messages are stored in the

collection. To ensure this doesn't happen, create an index on `authorId`.

```
Meteor.startup(function() {
   if (Meteor.isServer) {
       Messages._ensureIndex( { "authorId": 1 });
   }
});
```

You can confirm the index exists in your local database via Meteor mongo:

```
> meteor mongo
meteor:PRIMARY> use meteor
meteor:PRIMARY> db.Messages.getIndexes()
      {
            "v" : 1,
            "key" : {
                  "authorId" : 1
            },
            "name" : "type_1",
            "ns" : "meteor.Messages"
      }
```

Finally, to confirm that your query will take advantage of this index, use MongoDB's explain function to reveal how it will execute this query.

```
meteor:PRIMARY> db.Messages.find({authorId: "12345"}
).explain()
            "winningPlan" : {
                  "stage" : "FETCH",
                  "inputStage" : {
                        "stage" : "IXSCAN",
                        ...
```

In this case, MongoDB reports that when it executes this query, it will use the authorId index instead of spanning the entire collection to search for matching messages. This is indicated by the "`IXSCAN`" in the `winningPlan`'s `inputStage` stage value. If you see "`COLLSCAN`" instead, it means MongoDB did not find an index to help run the query and it will scan the entire collection — incurring a big performance hit.

WRITES

As usage grows, you also may notice database updates taking longer too. Just like the queries, updates will use indexes when available to find the documents to update. Thus, you should also create indexes to support update operations. For example, the following update:

```
Messages.update({ "authorId": authorId }, {$set: {archived:
true}}, {multi: true});
```

… will also also utilize the `authorId` index we created to quickly find and update all messages belonging to authorId.

When writing to the database, it's not just locating the documents to update that can slow things down. In MongoDB's original storage engine, MMAPv1. write operations to the same collection can only happen one at a time. Other concurrent writes would have to wait their turn — a situation that can quickly lead to bottlenecks.

Meteor 1.4 provides support for MongoDB's newest storage engine, called WiredTiger. WiredTiger offers document-level concurrency control. That means your application can perform concurrent write operations as long as they're updating different documents. The MongoDB team claims WiredTiger can deliver up to 7 - 10x performance improvement.

All new Meteor 1.4 applications use WiredTiger by default. If you have an existing application that you've upgraded to Meteor 1.4, and you don't mind losing your existing development data, simply type:

```
> meteor reset
```

… and your Meteor 1.4 app will create a new, WiredTiger-powered database the next time you start it up.

With your app running, you can confirm it's on WiredTiger with the mongo shell:

```
> meteor mongo
> db.serverStatus().storageEngine
{
    "name" : "wiredTiger",
    "supportsCommittedReads" : true,
    "persistent" : true
}
```

ISSUE #9: MINIMIZE SLOWDOWNS FROM THIRD-PARTY SERVICES

Sometimes the fault for bad performance lies elsewhere. Often our apps rely on data and services from third-party providers. And, while doing so allows us to leverage their platforms, the performance of our apps can be at their mercy if we're not careful.

Each time you utilize an external service, try to take into account:

- Any dependencies to the response of these services
- The expected performance of this service
- Any quotas or rate limits to this service

USING UNBLOCK()

Calling external services from the server are typically triggered by the client when it calls a Meteor method. Methods are processed in sequence for a client.

If a particular method is taking a long time because it's waiting for a response from an external service, subsequent methods will wait. This is because executing these methods in a predictable order is generally safer and leads to less bugs. From the user's point of view, however, this could slow down your app. If you're absolutely sure there are no dependencies on the response of this external service, then you can tell Meteor to go ahead and process subsequent method calls without waiting for the current one to complete.

For example, suppose we wanted to send an email notification to an offline user

when a new chat message is received. In this app, we can safely say that the sending of this email doesn't affect any other methods that may be running. Therefore we can tell Meteor that it's ok to process subsequent method calls from the same client using the unblock() function:

```
Meteor.methods({

   //notify the user a message was received
   notifyUserMessageReceived: function(username) {
       this.unblock(); // tell Meteor it's ok to execute
subsequent method calls
       EmailService.sendEmailNotification(username);
   }
});
```

CACHING RESPONSES

Sometimes we'll use an external service to query for data. If the data is not expected to change frequently – such as the current weather, or yesterday's sports scores – we can store the response on our app's server side and use this cache response for a given window of time. Only once we indicate that a cache is stale does the server have to query the third party service again.

The downside is that we're using up more server memory or local database storage, depending on how we're storing the response. It's important to calculate and take into account how big the cache can grow. However, the upside is we can potentially benefit from much faster performance, minimize any transaction costs, and avoid going over the usage quotas some external services may have.

ISSUE #10: USE HARDWARE ACCELERATION FOR MOBILE ANIMATIONS

The Project Ricochet team has found Meteor extremely useful for building mobile apps. It allows us to build apps for smartphones and tablets using the same familiar framework and assets that we already use for web apps. Meteor's Cordova integration makes this possible.

Mobile apps have come a long way. In many respects, they can provide better user experiences than their web app counterparts. With access to mobile-only hardware like GPS, accelerators, and fingerprint sensors, you could argue that mobile apps have raised the bar for what users expect from their technology interactions.

A clear example of that would be the animations that native apps often deploy to visually communicate feedback, information hierarchy, function changes, and prompts. Native apps have access to the device's GPU to generate these effects. In Meteor and Cordova, we can achieve similar animations using web technologies, like Javascript and CSS. However, because of the browser software they are rendered in, they often don't have the same silky smooth movements. Instead, what users see are choppy, clunky movements that distract us from the true intentions of these communication cues. Users also interpret this (rightly or wrongly) as slow app performance.

Mobile apps benefit from access to the device's graphical processing unit (GPU). The GPU hardware accelerates the rendering of graphics like animations. Fortunately, browsers today have also learned to utilize the GPU, particularly for 3D effects. In fact, we can even "trick" the browser to use the GPU to render 2D animations using a selected few CSS properties:

- transform
- opacity
- filter

To achieve smooth animations, use these properties when possible. For example, we can animate an object by modifying its `top` and `left` properties. This change doesn't get processed by the GPU.

The following will render a somewhat jittery animation:

```
.go-around-animation{
  animation: go-around-keyframes 10s infinite;
}
@keyframes go-around-keyframes{
  0%: {
    top: 0;
    left: 0;
  }

  25% {
    top: 0;
    left: 200px;
  }

  50% {
    top: 200px;
    left: 200px;
  }

  75% {
    top: 200px;
    left: 0;
  }
}
```

So, rather than modifying an object's `top` and `left` properties, use the `transform` property instead. You can achieve the same animation but have it rendered by the GPU:

```
.go-around-animation {
  animation: go-around-keyframes 10s infinite;
}

@keyframes go-around-keyframes{
  0%: {
    transform: translate(0, 0);
  }

  25% {
    transform: translate(200px, 0);
  }

  50% {
    transform: translate(200px, 200px);
  }
```

```
  75% {
    transform: translate(0, 200px);
  }
}
```

This also improves the app's performance on the desktop. Doing this for Meteor's Cordova apps ensures that your app can compete, in terms of user experience and the illusion of performance, with the other native apps that are also installed on the device.

EXPECT MORE CHALLENGES

Meteor is a full-stack framework that handles everything from the browser to the database and everything in between. As usage for your app grows, performance problems can surface at any point in the stack. Though Meteor makes it easy to achieve amazing real-time experiences, there are some potentially debilitating performance costs if you're not careful.

Editing the list down to 10 issues here was difficult. But, based on our experience, we saw these challenges most often. Likely, you'll run into a few of them in your apps too.

Hopefully these strategies will help you quickly fine tune your Meteor app. We want to see you quickly convert ideas into apps that perform spectacularly — and attract lots of users!

CONTACT US TO LEARN MORE

Questions about this white paper? Would you like to hear more about how Project Ricochet can help you implement a strategy that best leverages Marketing, Design, Content, and Engineering with regard to Drupal in your organization?

If so, please reach out via phone (800) 651-3186 or email info@projectricochet.com.