



AVIOX MAGAZINE

LLM APPS,
REAL-TIME VIDEO,
LOGGING AND
MULTI-TENANT
DESIGN

Engineering at Scale

VOL. 1 • NO. 3
SEPTEMBER 2026

In this issue

AI	3
Beyond the Chatbot: Engineering Context-Aware LLM Applications	3
RESEARCH	5
Peer-to-Peer by Design: How WebRTC Negotiates a Real-Time Connection	5
One Application, Many Tenants: Three Multi-Tenancy Architectures in Laravel	8
TECHNOLOGY	11
Centralized Logging with the ELK Stack, Containerized	11
DEVELOPMENT	15
GraphQL in React with Apollo Client: Queries, Mutations and Uploads	15

FROM THE EDITOR

Our September issue is about building systems that grow: more users, more data and more moving parts.

We open with what it takes to build LLM applications that understand context, not just answer questions. Then we look at how WebRTC sets up a real-time connection between two browsers, and compare three ways to serve many customers from one Laravel application. To keep all of it visible, we run the ELK stack in containers for central logging, and we finish with GraphQL in React using Apollo Client.

Thank you for reading. See you next month.

— *The Aviox Blogs Editors*

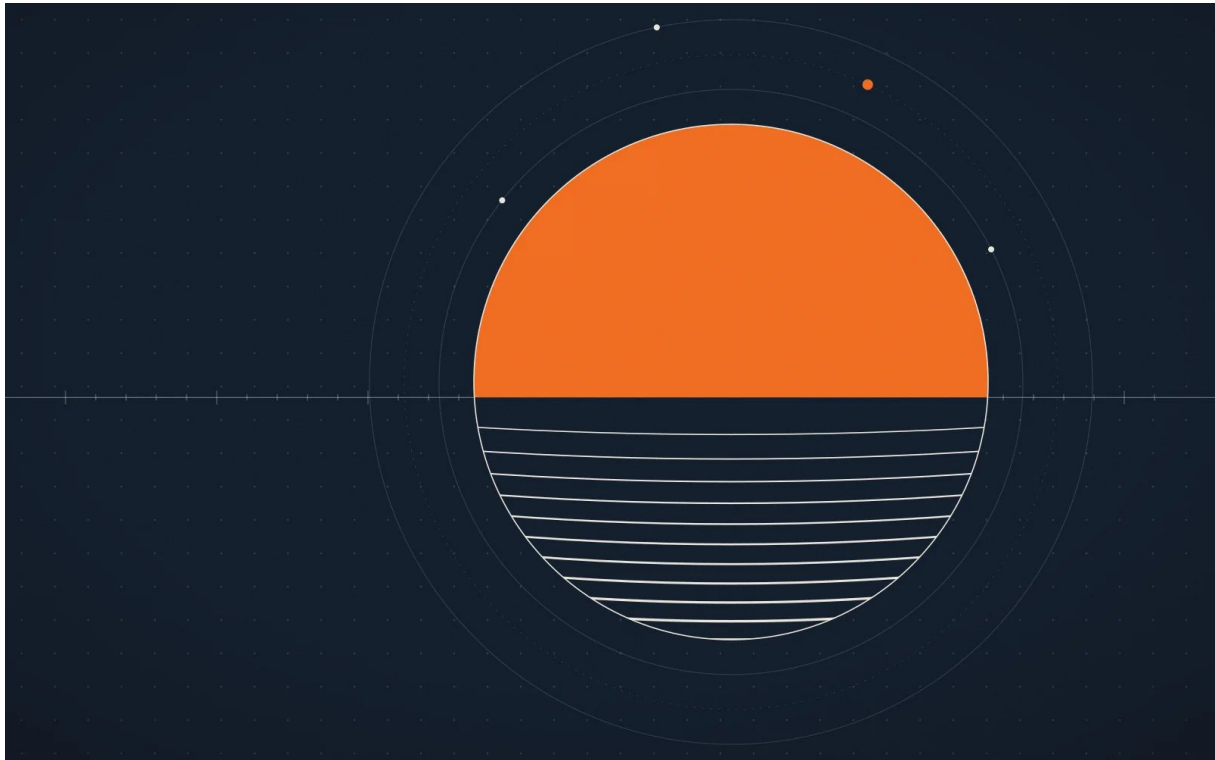
© 2026 Aviox Blogs. All rights reserved, except where noted: stories republished from other publishers remain theirs and are credited with their licence. No other part of this publication may be copied, changed or republished without written permission from Aviox Blogs.
Read this issue online at <https://blogs.avioxtechnologies.com/magazine/engineering-at-scale>

AI

Beyond the Chatbot: Engineering Context-Aware LLM Applications

The model is the easy part. What makes a language-model product useful is everything around it — the context it is given, the systems it can reach, and how rigorously it is tested.

By Aviox Blogs Editorial · 29 Sep 2026 · 3 min read



Almost every organization now has a chatbot prototype. It answers a few questions impressively in a demo, and then it meets real users — who ask about their own order, their own policy, their own account — and the illusion fades. The model never knew any of that. It was never given the context, and it was never connected to the systems where that context lives.

That gap is the real work of building with large language models. The interface may be a chat window, a voice assistant or an invisible step inside a workflow; what decides whether it is useful is the engineering around the model. At Aviox Blogs we break that work into four disciplines.

1. Start with intents, not prompts

The first instinct on an LLM project is to open a playground and start writing prompts. The better first step is quieter: understand the use cases, the intents behind them, and the workflows they sit inside. Which questions do people actually ask? What do they need to happen next? Where does an answer stop being enough and an action become necessary?

A few questions are worth answering before a single prompt is written:

- Which tasks should the assistant complete end to end, and which should it hand to a person?

- What information does it need to answer well — and where does that information live today?
- What must it never do, say or reveal?
- How will we know, in numbers, that it is helping?

The prompt is a specification. You cannot write a good one until you know what the system is for.

Aviox Blogs Editorial

2. Design the conversation, then the context

Conversational design is closer to interaction design than to copywriting. It means mapping dialogue flows and interaction paths, deciding how the system recovers when it misunderstands, and handling the multi-turn exchanges where meaning depends on what was said three messages ago.

Underneath the flow sits context modeling: deciding what the model should know at each turn. Too little, and it guesses. Too much, and it loses the thread or exposes information it shouldn't. Prototyping against real scenarios — not idealized ones — is the fastest way to find the balance.

Design for the unhappy path

Most of a conversational system's quality is decided in the moments it is unsure. Make uncertainty visible, ask a clarifying question, and offer a route to a person.

This story continues for Dispatch subscribers — about 2 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

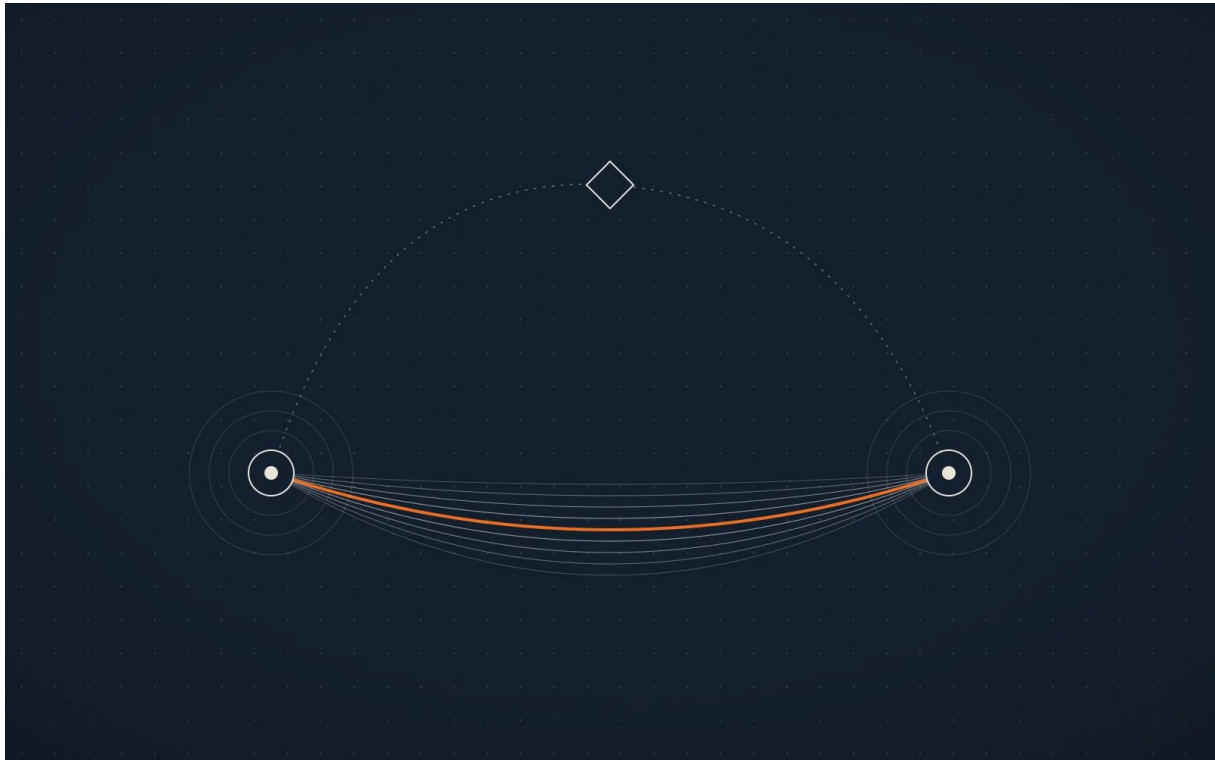
Read online: <https://blogs.avioxtechnologies.com/story/beyond-the-chatbot-engineering-context-aware-llm-applications>

RESEARCH

Peer-to-Peer by Design: How WebRTC Negotiates a Real-Time Connection

A close reading of the steps between “start call” and the first video frame — media capture, signaling, session description and the ICE machinery that traverses NAT.

By Aviox Blogs Engineering · 24 Sep 2026 · 7 min read



Introduction

WebRTC (Web Real-Time Communications) is an open-source project and set of browser APIs that enable real-time voice, video and data exchange between browsers and devices. Developers use JavaScript APIs to create peer-to-peer connections, without custom interfaces, extra plugins or special software. Opening a web page is enough.

The simplicity of that experience hides a surprisingly intricate negotiation. Two devices, usually behind different routers, must discover how to reach each other, agree on codecs and formats, and establish an encrypted channel — all in the second or two between a click and a picture.

Architecture at a glance

Diagram: Peer A and Peer B exchange offers, answers and ICE candidates through a signaling server; each asks a STUN server for its public address; media flows directly between the peers, or through a TURN relay when a direct path is blocked.

Signaling carries descriptions; STUN discovers addresses; TURN relays when nothing else works.

Four concerns are layered here. **Media capture** obtains streams from cameras, microphones or the screen. **Signaling** carries metadata between peers. **Session description** tells each side what the other can send and receive. **Connectivity establishment** finds a network path. We take them in order.

Media capture

Cameras and microphones are reached through `navigator.mediaDevices`. The most common entry point is `getUserMedia()`, which takes a constraints object and resolves to a `MediaStream`. Calling it triggers a permission prompt; denial raises a `NotAllowedError`, and a missing device a `NotFoundError`.

```
const constraints = { video: true, audio: true };

try {
  const stream = await navigator.mediaDevices.getUserMedia(constraints);
  document.querySelector("video#local").srcObject = stream;
} catch (error) {
  console.error("Could not open media devices", error);
}
```

Real applications enumerate devices so users can choose, and listen for devices being connected or removed during a session:

```
async function listDevices(kind) {
  const devices = await navigator.mediaDevices.enumerateDevices();
  return devices.filter((device) => device.kind === kind);
}

navigator.mediaDevices.addEventListener("devicechange", async () => {
  renderCameraPicker(await listDevices("videoinput"));
});
```

Constraints

Constraints may be loose (“any camera”) or precise — an exact `deviceId`, a minimum resolution, echo cancellation on. Values can be specific, ranges, or objects with `ideal` and `exact` keys; with `exact`, only matching streams are returned. Detailed constraints produce better streams than defaults, and `applyConstraints()` re-configures a live track without reopening the device.

```
const stream = await navigator.mediaDevices.getUserMedia({
  audio: { echoCancellation: true },
  video: {
    deviceId: { exact: cameraId },
    width: { min: 640, ideal: 1280 },
    height: { min: 480, ideal: 720 },
  },
});

const [track] = stream.getVideoTracks();
console.log(track.getSettings()); // what was actually applied
```

Screen capture uses `getDisplayMedia()`, which returns a `MediaStream` that behaves identically but accepts its own constraint set. For local playback, a `video` element should carry `autoplay` and `playsinline` so streams start immediately and play inline on mobile browsers.

This story continues for Dispatch subscribers — about 4 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

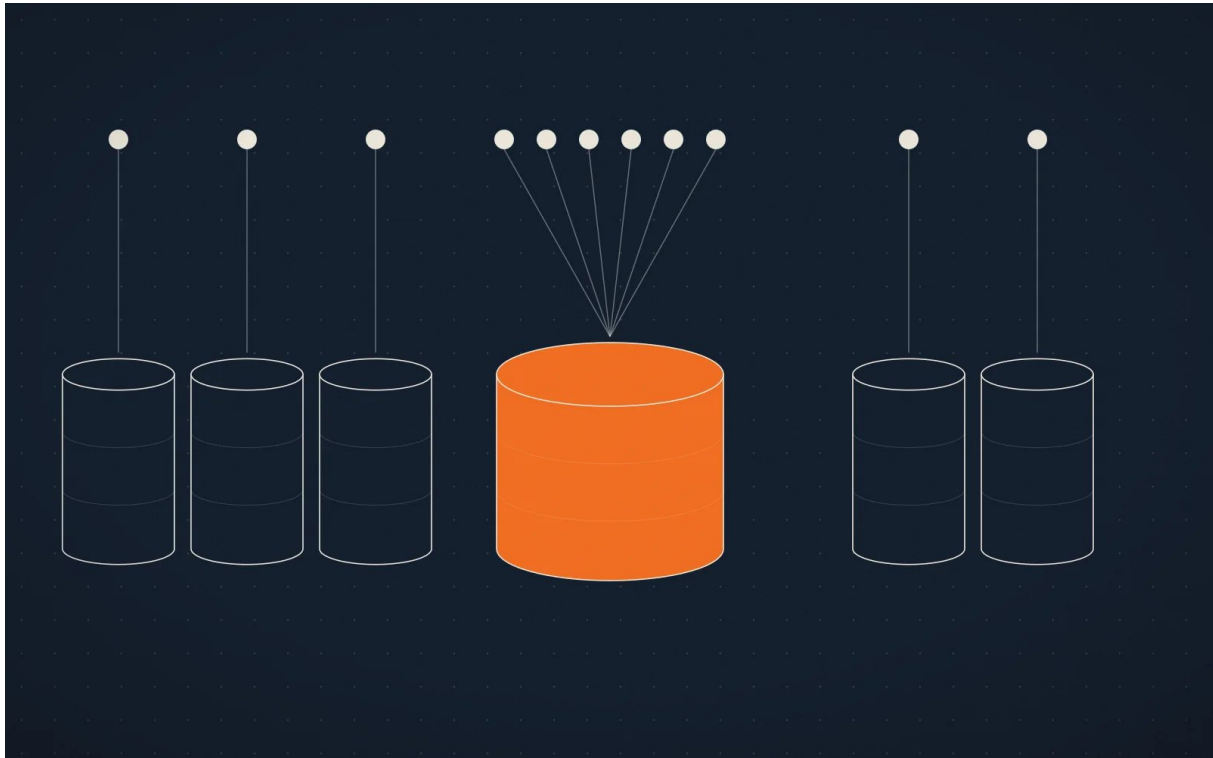
Read online: <https://blogs.avioxtechnologies.com/story/webrtc-peer-to-peer-by-design>

RESEARCH

One Application, Many Tenants: Three Multi-Tenancy Architectures in Laravel

Database per tenant, a shared schema with tenant identifiers, or a dedicated package. A comparative study of isolation, cost and operational complexity.

By Aviox Blogs Engineering · 3 Sep 2026 · 5 min read



Introduction

Multi-tenancy is an architectural approach that allows a single application to serve multiple clients — tenants — while maintaining data isolation and allowing per-tenant customization. In a Laravel application, it requires deliberate design: where tenant data lives, how each request discovers its tenant, and how every query is kept within the right boundary.

Diagram comparing three tenancy models: separate databases with one per tenant; a single shared database whose rows carry a `tenant_id` column; and a package layer that manages tenant resolution and databases on the application's behalf.

Three models, trading isolation against operational simplicity.

Approach 1: A separate database per tenant

Each tenant receives its own database, giving complete data segregation. This suits tenants that require strong isolation or need to scale independently.

Tenant registry

A central table records each tenant and the database that holds its data:

```
public function up(): void
{
    Schema::create('tenants', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('domain')->unique();
        $table->string('database')->unique();
        $table->timestamps();
    });
}
```

database/migrations/xxxx_xx_xx_create_tenants_table.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Tenant extends Model
{
    protected $fillable = ['name', 'domain', 'database'];
}
```

app/Models/Tenant.php

Switching connections per request

Middleware resolves the tenant from the request's host and points the connection at that tenant's database before any query runs:

```
namespace App\Http\Middleware;

use App\Models\Tenant;
use Closure;
use Illuminate\Support\Facades\DB;

class TenantMiddleware
{
    public function handle($request, Closure $next)
    {
        $tenant = Tenant::where('domain', $request->getHost())->firstOrFail();

        config(['database.connections.mysql.database' => $tenant->database]);
        DB::purge('mysql');
        DB::reconnect('mysql');

        return $next($request);
    }
}
```

app/Http/Middleware/TenantMiddleware.php

Apply it to every route that serves tenant data:

```
Route::middleware(TenantMiddleware::class)->group(function () {
    Route::get('/dashboard', DashboardController::class);
    // ...all tenant routes
});
```

routes/web.php

Keep the registry separate

In production the tenants table usually lives on its own central connection, so tenant resolution never depends on the database being switched to.

This story continues for Dispatch subscribers — about 3 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

Read online: <https://blogs.avioxtechnologies.com/story/multi-tenancy-in-laravel-three-architectures>

TECHNOLOGY

Centralized Logging with the ELK Stack, Containerized

Logs scattered across servers answer no questions. Elasticsearch, Logstash and Kibana turn them into one searchable record — and Docker Compose makes the stack reproducible.

By Aviox Blogs Engineering · 17 Sep 2026 · 7 min read



Every application produces logs, and most teams only read them in a crisis — over SSH, one server at a time. The ELK stack changes that. Elasticsearch stores and indexes events, Logstash collects and shapes them, and Kibana lets anyone filter and explore them in a browser.

This guide introduces each component, then runs the whole stack in Docker with security enabled from the first boot.

The three components

Diagram: application logs flow into Logstash, which runs input, filter and output stages before writing to Elasticsearch; Kibana reads from Elasticsearch to search and visualise.

Logstash shapes events, Elasticsearch indexes them, Kibana makes them readable.

Elasticsearch

A distributed engine for real-time search and analysis. It indexes data according to its type, which makes retrieval very fast. On first start it configures security automatically — generating certificates, a password for the `elastic` user and an enrolment token for Kibana. Keep them somewhere safe.

Logstash

The collection engine. It gathers the data your applications produce, transforms it, and stores it in Elasticsearch. The simplest route is to keep logging to files as you already do and let Logstash read them.

Kibana

The window onto everything Elasticsearch holds. On first launch it prints a one-time link; open it, paste the enrolment token, then sign in as `elastic`.

Anatomy of a pipeline

A Logstash pipeline is a `.conf` file with three sections:

- **Input** — where events come from. Input plugins read from a specific source, and codecs decode formats such as CSV on the way in.
- **Filter** — intermediary processing that makes events easier to index. Filters are often applied conditionally, depending on the characteristics of the event.
- **Output** — where events go. Here, the Elasticsearch plugin writes each event to an index you choose.

You can run several pipelines side by side by declaring them in `pipelines.yml`.

Running the stack in Docker

If your application already uses Docker Compose, merge the services below into your existing file; otherwise create a new one. The finished project looks like this:

```
.
+-- docker-compose.yml
+-- .env
+-- elasticsearch/
|   +-- config/elasticsearch.yml
|   +-- Dockerfile
+-- kibana/
|   +-- config/kibana.yml
|   +-- Dockerfile
+-- logstash/
|   +-- config/logstash.yml
|   +-- pipeline/logstash.conf
|   +-- Dockerfile
+-- setup/
    +-- Dockerfile
    +-- entrypoint.sh
    +-- lib.sh
    +-- roles/logstash_writer.json

services:
  # One-off job: creates the logstash_internal and kibana_system users
  # with the passwords from .env, then exits on every later run.
  setup:
    build:
      context: setup/
      args:
        ELASTIC_VERSION: ${ELASTIC_VERSION}
    init: true
    volumes:
      - ./setup/entrypoint.sh:/entrypoint.sh:ro,Z
      - ./setup/lib.sh:/lib.sh:ro,Z
      - ./setup/roles:/roles:ro,Z
      - setup:/state:Z
    environment:
      ELASTIC_PASSWORD: ${ELASTIC_PASSWORD:-}
```

```

    LOGSTASH_INTERNAL_PASSWORD: \${LOGSTASH_INTERNAL_PASSWORD:-}
    KIBANA_SYSTEM_PASSWORD: \${KIBANA_SYSTEM_PASSWORD:-}
  networks: [elk]
  depends_on: [elasticsearch]

elasticsearch:
  build:
    context: elasticsearch/
    args:
      ELASTIC_VERSION: \${ELASTIC_VERSION}
  volumes:
    - ./elasticsearch/config/elasticsearch.yml:/usr/share/elasticsearch/config/
elasticsearch.yml:ro,Z
    - elasticsearch:/usr/share/elasticsearch/data:Z
  ports: ["9200:9200", "9300:9300"]
  environment:
    node.name: elasticsearch
    ES_JAVA_OPTS: -Xms512m -Xmx512m
    ELASTIC_PASSWORD: \${ELASTIC_PASSWORD:-}
    discovery.type: single-node
  networks: [elk]
  restart: unless-stopped

logstash:
  build:
    context: logstash/
    args:
      ELASTIC_VERSION: \${ELASTIC_VERSION}
  volumes:
    - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro,Z
    - ./logstash/pipeline:/usr/share/logstash/pipeline:ro,Z
  ports: ["5044:5044", "50000:50000/tcp", "50000:50000/udp", "9600:9600"]
  environment:
    LS_JAVA_OPTS: -Xms256m -Xmx256m
    LOGSTASH_INTERNAL_PASSWORD: \${LOGSTASH_INTERNAL_PASSWORD:-}
  networks: [elk]
  depends_on: [elasticsearch]
  restart: unless-stopped

kibana:
  build:
    context: kibana/
    args:
      ELASTIC_VERSION: \${ELASTIC_VERSION}
  volumes:
    - ./kibana/config/kibana.yml:/usr/share/kibana/config/kibana.yml:ro,Z
  ports: ["5601:5601"]
  environment:
    KIBANA_SYSTEM_PASSWORD: \${KIBANA_SYSTEM_PASSWORD:-}
  networks: [elk]
  depends_on: [elasticsearch]
  restart: unless-stopped

networks:
  elk:
    driver: bridge

volumes:
  setup:
  elasticsearch:

```

docker-compose.yml

image or build?

A service can either pull a ready-made image or build from its own Dockerfile. Building each component separately makes it easy to add plugins to Elasticsearch, Kibana or Logstash later without touching the others.

Each component's Dockerfile simply pins the official image to the version in `.env` — the place to add plugins later:

```
ARG ELASTIC_VERSION
FROM docker.elastic.co/logstash/logstash:${ELASTIC_VERSION}

# Add Logstash plugins here, for example:
# RUN logstash-plugin install logstash-filter-json
```

logstash/Dockerfile

The pipeline itself listens on TCP port 50000 and writes everything to Elasticsearch as the restricted `logstash_internal` user:

```
input {
  tcp {
    port => 50000
  }
}

## Add filters / plugin configuration here

output {
  elasticsearch {
    hosts => "elasticsearch:9200"
    user => "logstash_internal"
    password => "${LOGSTASH_INTERNAL_PASSWORD}"
  }
}
```

logstash/pipeline/logstash.conf

Versions and credentials live in one place. The `setup` service reads them on first boot to create users and roles; to add Filebeat or Metricbeat later, declare their passwords in `entrypoint.sh` and add a matching role file under `setup/roles`.

```
ELASTIC_VERSION=8.6.2

# Built-in superuser
ELASTIC_PASSWORD='changeme'

# Custom user Logstash uses to write to Elasticsearch
LOGSTASH_INTERNAL_PASSWORD='changeme'

# Built-in user Kibana uses to talk to Elasticsearch
KIBANA_SYSTEM_PASSWORD='changeme'

.env
```

This story continues for Dispatch subscribers — about 1 more minute of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

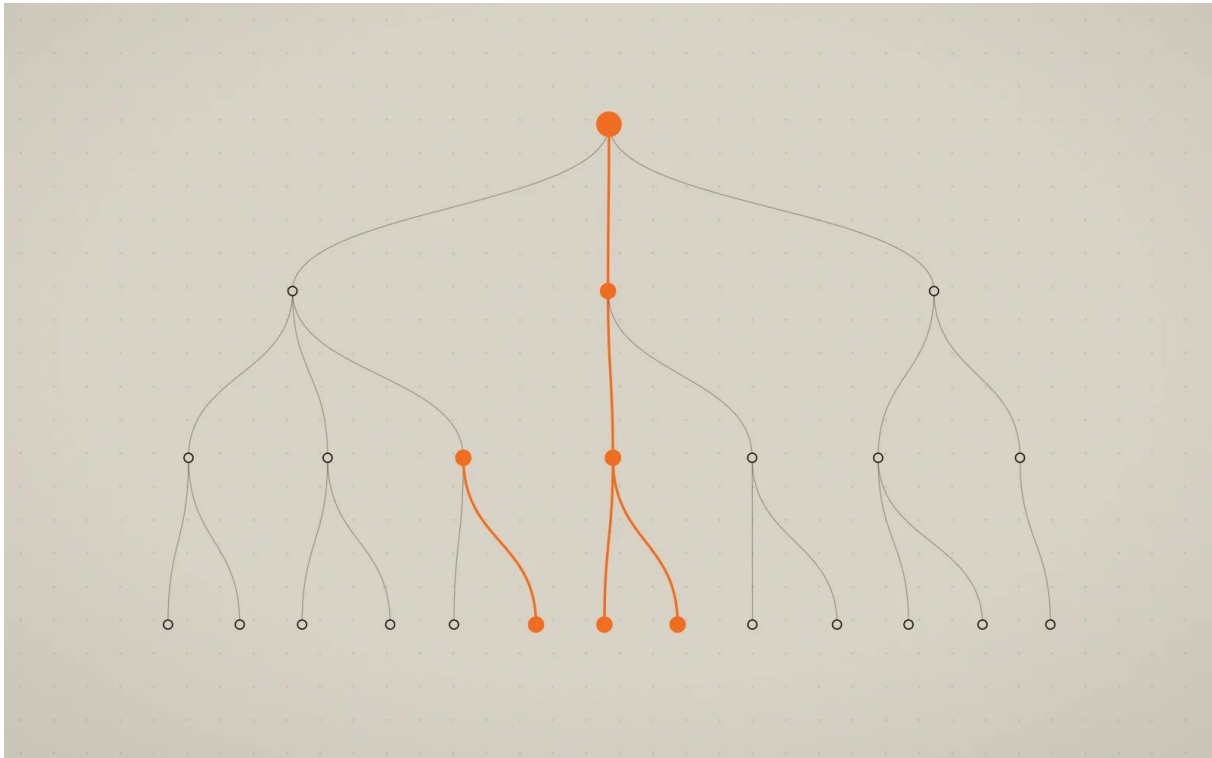
Read online: <https://blogs.avioxtechnologies.com/story/centralized-logging-with-the-elk-stack>

DEVELOPMENT

GraphQL in React with Apollo Client: Queries, Mutations and Uploads

Ask for exactly the data a screen needs, in one request, and let a normalized cache keep the interface in sync. A practical walkthrough of Apollo Client in React.

By Aviox Blogs Engineering · 10 Sep 2026 · 6 min read



Why GraphQL

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. A client describes the shape of the data it wants and receives exactly that — nothing more, nothing less — in a predictable structure.

Queries can follow references between resources, so data that a typical REST API would spread across several URLs arrives in a single request. Because the client controls what it fetches, applications stay fast and stable as screens evolve.

What Apollo Client adds

Apollo Client is a fully featured JavaScript GraphQL client. It sends queries and mutations, caches results in a normalized in-memory store, tracks loading and error states, and integrates with React, Angular and Vue. Its cache also doubles as a home for local state, kept in step with the server.

Install it alongside the reference `graphql` package:

```
npm install @apollo/client graphql
```

@apollo/client

Querying and mutating data, a normalized cache for local and remote data, subscriptions for real-time updates, error handling and retries, and links for customizing requests.

graphql

The reference implementation used to parse and validate queries. On the server it defines schemas and resolvers; on the client it parses the documents you send.

Connecting the client

Create one client for the application — pointing an `HttpLink` at your GraphQL endpoint and giving it a cache — then provide it to the component tree:

```
import { createRoot } from "react-dom/client";
import { ApolloClient, HttpLink, InMemoryCache } from "@apollo/client";
import { ApolloProvider } from "@apollo/client/react";
import App from "../App";

const client = new ApolloClient({
  link: new HttpLink({ uri: "https://api.example.com/graphql" }),
  cache: new InMemoryCache(),
});

createRoot(document.getElementById("root")).render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
);
```

src/main.jsx

Version note

Since Apollo Client 4, React APIs such as `ApolloProvider`, `useQuery` and `useMutation` are imported from `@apollo/client/react`, and the endpoint is configured through a `link` rather than a `uri` shortcut. Older projects on version 3 import everything from `@apollo/client`.

Reading data with useQuery

Define the query with `gql`. Its structure mirrors the data you expect back — one of GraphQL's defining features:

```
import { gql } from "@apollo/client";

export const GET_USERS = gql`
  query GetUsers {
    users {
      id
      name
      email
    }
  }
`;
```

src/queries.js

Arguments narrow the result. To fetch only the name and email of user 123, ask for exactly that:

```
query GetUser {
  user(id: 123) {
```

```
      name
      email
    }
  }
}
```

The `useQuery` hook runs the query, re-renders as its state changes, and serves repeat requests from the cache:

```
import { useQuery } from "@apollo/client/react";
import { GET_USERS } from "../queries";

export default function UserList() {
  const { loading, error, data } = useQuery(GET_USERS);

  if (loading) return <p>Loading...</p>;
  if (error) return <p role="alert">Couldn't load users: {error.message}</p>;

  return (
    <ul>
      {data.users.map((user) => (
        <li key={user.id}>
          {user.name} ({user.email})
        </li>
      ))}
    </ul>
  );
}
```

src/UserList.jsx

This story continues for Dispatch subscribers — about 3 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

Read online: <https://blogs.avioxtechnologies.com/story/graphql-in-react-with-apollo-client>