Mentiq — Frontend SDK: Plan, Flow, and Feature Roadmap

This document explains, in simple words, how the frontend SDK should work, how to build it feature-by-feature, and how to deploy it. It focuses on delivering one complete feature at a time so you can ship working parts quickly.

1) What the SDK does (simple)

The SDK is a small JavaScript library that web apps include. It does three main things:

• Capture actions users do (page views, clicks, form submits).

• Group events into short sessions and attach a user id when available.

• Send events to your backend in batches for analytics and further processing.

2) High-level architecture (simple)

Think of the system as two halves:

A. Frontend: the SDK (this is your area). It runs in the customer's browser and captures events.

B. Backend: the collector and processing pipeline (the backend team handles this). The SDK posts events to the collector.

How data flows:

1. SDK captures events and stores them in a small queue in browser memory.

2. SDK batches and posts events to the collector endpoint (e.g., /collect).

3. Collector accepts and stores events to a queue (e.g., Kafka) or DB for processing.

4. Downstream services (analytics DB + replay storage) process events for dashboards and replays.

3) Distribution: how customers add the SDK

We provide two ways to add the SDK:

• npm package (for developers who use modern build systems). They `npm install` and import the SDK.

• script-tag (a single small JS file hosted on CDN). Users paste a `` into their HTML.

Start with the script-tag prototype (fast to demo). Then convert the same code into an npm package.

4) Feature-by-feature plan (build one complete feature, then next)

We will use a feature-first approach. Implement and polish one feature fully before moving to the next. Each feature includes: code, tests, docs, demo page, and a small checklist.

Feature 1 — Core event tracking (Complete this first)

Goal: A working SDK that can capture `page_view`, `track(event)`, and `identify(userId)` and send them to the collector.

Why: This covers the basic needs for dashboards and makes a useful demo quickly.

What to implement (step-by-step):

1. Public API:

• init(config) — set collectUrl and apiKey (optional).

• track(eventName, properties) — record an event.

• identify(userId) — attach a user id to future events.

• flush() — force send queued events.

2. Queue & batching:

• Keep an array of events in memory.

• Send when array length >= MAX_BATCH (e.g., 10) or every FLUSH_MS (e.g., 3000ms).

3. Sending logic:

• Use navigator.sendBeacon on page unload if available.

• Otherwise use fetch() with `keepalive` where supported.

• Retry a failed batch by putting it back into the queue (simple retry).

4. Session ID:

• Create a session id stored in sessionStorage to group events in one visit.

5. Demo page:

• A simple HTML page with a few buttons (data-mentiq-track attributes) and a login simulation to call `identify()`.

6. Tests / validation:

• Run the demo page and show collector receiving events. Check the browser console logs and network requests.

Acceptance criteria (Done when):

• Demo page shows real-time events in the collector terminal.

• `identify()` attaches user id in the events.

• Batch and flush behaviors work (after 3s or 10 events).

Feature 2 — Auto-capture for marked clicks and forms

Goal: Automatically capture clicks and form submissions only for elements that you mark.

What to implement:

1. Use a `data-mentiq-track` attribute on elements to mark them for tracking.

2. Listen to `click` events on the document and locate the nearest marked ancestor.

3. Capture small useful properties: element text, href (if link), and a simple selector for identification.

4. For forms, listen to `submit` and send `form_submit` with success/failure if possible.

Acceptance criteria:

• Buttons with `data-mentiq-track` send events when clicked.

• Console logs show queued events. Collector receives them in batches.

Feature 3 — SPA route changes and page views

Goal: Make the SDK work correctly in single-page apps (React/Next/Vue).

What to implement:

1. Detect route changes (use `history.pushState`/`popstate` or allow manual `page()` calls).

2. Emit `page_view` on route change with the new path and title.

3. Provide a helper for frameworks (small docs showing how to call `mentiq.page()` on route change).

Acceptance criteria:

• Page views are tracked when route changes in SPA demo.

Feature 4 — Basic privacy & sampling controls

Goal: Respect privacy and keep data volume manageable.

What to implement:

1. Respect `window.doNotTrack` by default (skip sending if set).
2. Mask form inputs by default (do not capture sensitive input values).
3. Add a `sampleRate` config option to record only a percentage of sessions (e.g., 0.1 = 10%).

Acceptance criteria:

• Masked inputs are not included in event properties.

• Sampling correctly reduces event volume for demo runs.

Feature 5 — Session Replay (optional, add after core)

Goal: Record a user's session (DOM snapshots, inputs) and upload them to a replay endpoint for playback.

Important: This is heavier work and should be optional and sampled.

What to implement:

1. Use rrweb and load it dynamically when `enableReplay` is set to true in config.
2. Record events and send replay chunks separately from analytics events (to a replay-specific endpoint).
3. Always mask inputs by default and provide configuration for selectors to whitelist/blacklist.

Acceptance criteria:

• Replay chunks reach backend; small playback demo can show recorded events.

5) Implementation details (developer notes) — simple

Which language and bundler:

• TypeScript for the code (better developer experience and types).

• Build with esbuild/tsup/rollup to produce ESM (module) and UMD (for CDN) bundles.

Code structure (suggested):

• src/index.ts — public API (init, track, identify, page, flush).

• src/network.ts — send/flush helper code (fetch + sendBeacon).

• src/autoCapture.ts — click/form capture logic.

• src/replay.ts — code to lazy-load rrweb and send replay blobs (optional).

• dist/mentiq.umd.js — UMD build for CDN script.

• package.json — exports for ESM and CJS for npm.

6) Testing & QA (simple steps)

1. Local collector: a small Express server that echoes and logs events. Use it to validate events.
2. Unit tests: test queueing, batching, and flush retry logic.
3. E2E: a demo page in a simple app (static HTML + SPA example) to test identify, track, page changes, and sendBeacon.
4. Manual privacy tests: verify inputs are masked and doNotTrack respected.

7) Deployment & release

Script-tag (fast):

• Host the UMD bundle (dist/mentiq.umd.js) on your CDN or a static hosting (S3 + CloudFront).

• Provide customers the small loader snippet that queues calls and loads the UMD file.

NPM package:

• Prepare package.json with `module`, `main`, and `types` fields.

• Publish to npm (npm publish) with semantic versioning (v0.1.0).

• Provide usage docs for React/Next.js (how to import and init).

8) Monitoring and metrics for the SDK

Track SDK performance and errors so you don't break customer sites:

1. SDK size — ensure gzipped size is small (aim < 40KB for core).
2. Error logging — capture and ship errors from the SDK to your internal error tracker.
3. Delivery success rate — monitor how many events are delivered vs queued/retried.
4. CPU / memory — avoid heavy operations on the main thread (lazy-load heavy libraries).

9) Roadmap & timeline (realistic)

Day 0 (Today) — Prototype script and demo page (done).

Day 1 — Turn prototype into an npm/UMD basic SDK (core feature complete: track/identify/page_view/batching).

Day 2 — Auto-capture clicks & SPA support + tests.

Day 3 — Privacy controls, sampling, and documentation.

Day 4-7 — Replay (if needed), polish, publish to npm, and CDN hosting.

Adjust timeline to your team size and priorities. The goal: ship a small, solid core first.

10) Checklist before you call it 'done' for MVP

• SDK core API implemented and documented (init, track, identify, page, flush).

• Demo page that shows events arriving at collector live.

• Auto-click capture for marked elements.

• Session id works and identify attaches user id.

• sendBeacon and beforeunload handling implemented.

• Basic privacy defaults (mask inputs, respect doNotTrack).

• npm package ready and UMD bundle hosted on CDN (or at least built).

Appendix: Quick examples (how to use the SDK) — copy/paste

1) Script-tag (minimal loader + config):

```
window.__MENTIQ_CONFIG={collectUrl:'https://api.yourapp.com/collect',apiKey:'prod_key'};
```

2) Simple calls from the page:

__mentiq.identify('user_123'); __mentiq.track('signup',{plan:'trial'});

Good luck — if you want, I can now generate:

• The exact starter repository files (src code + build script) ready to copy, or

• The UMD loader snippet optimized and minified for distribution, or

• The TypeScript SDK `src/index.ts` file with comments and tests.

Tell me which of those you'd like next and I will produce it.