Real-time Web Applications Course

Flavio Copes

Updated August 3, 2026

Contents

Build reliable live interfaces with polling, server-sent events, WebSockets, snapshots, reconnection, backpressure, and graceful degradation.

What you'll learn: Build and operate a live service-status board that remains correct through disconnects, duplicates, restarts, slow clients, and fallback mode.

Choose the channel

Turn “real-time” into requirements and choose polling, SSE, or WebSockets deliberately.

Define what real-time means

Turn a vague demand for instant updates into measurable latency, direction, ordering, and recovery requirements.

Before choosing a tool or writing more code, make the requirement concrete. A status page that may update within five seconds needs a different design from a collaborative cursor that should move within a fraction of a second.

Real-time is a product requirement with a latency budget, not the name of one protocol. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to declare “use WebSockets” before measuring direction, frequency, audience, and acceptable delay. The happy path may still work, which makes the mistake easy to miss. write the update direction, latency budget, message rate, audience size, ordering rule, and recovery requirement first. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Choose the least complicated channel that meets the product requirement, and keep a non-live path for recovery. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Write the requirements for incident creation, incident updates, viewer counts, and operator acknowledgements. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Start with deliberate polling

Use conditional HTTP polling when updates are infrequent and a short delay is acceptable.

Start from the behavior you can observe. The board can request an incident snapshot every fifteen seconds and use validators so unchanged responses stay small.

Polling remains a good design when requests are cheap, updates are rare, and a bounded delay is acceptable. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to dismiss polling as primitive and replace one predictable request with a permanent connection immediately. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to implement a measured polling baseline with cancellation, backoff, visibility awareness, and HTTP caching. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Choose the least complicated channel that meets the product requirement, and keep a non-live path for recovery. Record enough evidence that another developer can repeat the result without relying on your memory.

Build the snapshot endpoint and record request count, bytes transferred, and worst-case staleness over two minutes. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Compare SSE and WebSockets

Choose server-sent events for one-way streams and WebSockets when both peers must send frequent messages.

Incident updates naturally flow from server to many browsers, while a shared drawing surface sends frequent messages in both directions. That contrast gives us something useful to test instead of a rule to memorize.

SSE is an HTTP response that carries server-to-browser events; WebSockets create a full-duplex message channel after an HTTP upgrade. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can choose by popularity instead of message direction, deployment support, and reconnection behavior and still get one successful demo. Push past the demo. draw the actual message arrows and select SSE or WebSockets from those arrows and the operating constraints, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Choose the least complicated channel that meets the product requirement, and keep a non-live path for recovery. Save the before-and-after evidence now; it will make the final project review much more honest.

Sketch both architectures for the status board, including authentication, proxies, reconnects, and a snapshot request. Save the evidence, then explain which requirement would force you to choose a different design.

Keep HTTP as the foundation

Use normal requests for initial state and commands, then reserve the live channel for changes.

The board loads a complete incident snapshot over HTTP, submits operator commands over HTTP, and streams subsequent changes. This is a small part of a live service-status board that streams incidents without losing its ordinary HTTP foundation, but the boundary it creates affects everything that follows.

A live connection should complement HTTP rather than replace every request in the application. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to send the entire application protocol through one long-lived socket because it feels more real-time. It makes later failures harder to locate. Instead, separate snapshots, commands, and event notifications so each has an inspectable contract and retry policy. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Choose the least complicated channel that meets the product requirement, and keep a non-live path for recovery. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Define one snapshot route, one command route, and one event envelope; explain which one repairs a missed update. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: choose the channel

Check polling, streaming, direction, latency, connection cost, and the role of ordinary HTTP.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build an SSE stream

Frame, name, resume, heartbeat, and clean up a one-way event stream.

Frame an event stream

Return the correct content type and encode events as lines separated by a blank line.

Before choosing a tool or writing more code, make the requirement concrete. The status server writes one compact JSON object in a `data:` field rather than concatenating unframed JSON values.

An SSE response uses `text/event-stream`; fields such as `event`, `id`, `data`, and `retry` are line based, and a blank line dispatches one event. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to write arbitrary text to a long response and assume EventSource can discover the boundaries. The happy path may still work, which makes the mistake easy to miss. set the event-stream headers and serialize every event with explicit fields and a terminating blank line. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Build an SSE feed that reconnects cleanly and can repair a gap instead of merely looking live in one browser tab. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Create a tiny SSE endpoint, inspect its raw response with curl, and verify that two events arrive as two browser events. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Name and version events

Design event names and payloads that can evolve without making every client guess what changed.

Start from the behavior you can observe. The board emits `incident.created` and `incident.updated` with an event id and a payload version instead of a generic `message` blob.

A live feed is an API. Stable names, identifiers, timestamps, and versioned payload shapes make it debuggable. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to mirror internal database rows directly into the public stream. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to define a small event envelope with a stable type, id, occurred-at time, version, and documented payload. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Build an SSE feed that reconnects cleanly and can repair a gap instead of merely looking live in one browser tab. Record enough evidence that another developer can repeat the result without relying on your memory.

Write three event examples and show how an older client ignores a new optional field safely. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Resume after a disconnect

Use event identifiers and a snapshot fallback so reconnection cannot silently create a gap.

If incident 42 was the last applied event, the server replays later retained events or tells the client to fetch a fresh snapshot. That contrast gives us something useful to test instead of a rule to memorize.

EventSource reconnects automatically and can send `Last-Event-ID`, but the server must decide how much history it can replay. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can treat reconnection as success even when the server has already discarded the missing events and still get one successful demo. Push past the demo. retain a bounded event log and fall back to a fresh snapshot when the requested position is no longer available, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Build an SSE feed that reconnects cleanly and can repair a gap instead of merely looking live in one browser tab. Save the before-and-after evidence now; it will make the final project review much more honest.

Disconnect the browser, create several incidents, reconnect, and prove the final state matches a fresh snapshot. Save the evidence, then explain which requirement would force you to choose a different design.

Heartbeat and clean up

Keep intermediaries aware of quiet streams and release resources as soon as a client leaves.

A comment heartbeat keeps the connection active while an abort signal removes the subscriber from the server set. This is a small part of a live service-status board that streams incidents without losing its ordinary HTTP foundation, but the boundary it creates affects everything that follows.

Long-lived responses cross browsers, proxies, servers, and load balancers, each with timeouts and resource limits. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to create an interval per client and never clear it after navigation or network loss. It makes later failures harder to locate. Instead, send a modest heartbeat, observe the request abort signal, and remove timers and subscriptions during cleanup. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Build an SSE feed that reconnects cleanly and can repair a gap instead of merely looking live in one browser tab. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Open and close twenty clients while tracking active connections, timers, and memory; the counts must return to baseline. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: build an sse stream

Check event-stream framing, named events, identifiers, reconnects, heartbeats, and cleanup.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build a WebSocket channel

Handle upgrades, messages, authorization, and bounded flow.

Follow the WebSocket upgrade

Trace the HTTP handshake into a persistent ws or wss connection and understand what infrastructure must support.

Before choosing a tool or writing more code, make the requirement concrete. The operator console uses `wss:` in production, and the reverse proxy must preserve the upgrade headers and idle connection.

A WebSocket begins as an HTTP request asking to upgrade protocols, then becomes a framed bidirectional connection. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to debug only the JavaScript client when a proxy or platform rejects the upgrade. The happy path may still work, which makes the mistake easy to miss. inspect the handshake, negotiated protocol, response status, proxy configuration, and TLS boundary as one path. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Build a bidirectional operator channel with explicit messages, bounded queues, and observable connection state. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Capture a successful and failed handshake, then identify the exact layer that changed the result. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Design message envelopes

Give every WebSocket message a type, identifier, version, and validation path.

Start from the behavior you can observe. An acknowledgement references the client command id, while an incident update carries a server event id used for deduplication.

WebSockets transport messages; they do not define the meaning, authorization, or compatibility of those messages. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to switch on loosely shaped JSON and trust every field because the connection is already open. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to parse safely, validate against a documented envelope, reject unknown types, and correlate commands with acknowledgements. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Build a bidirectional operator channel with explicit messages, bounded queues, and observable connection state. Record enough evidence that another developer can repeat the result without relying on your memory.

Define command, acknowledgement, event, and error envelopes and test malformed JSON and an unknown message version. Repeat it once with an invalid or hostile condition and write down the

boundary the failure revealed.

Authenticate and authorize connections

Authenticate the handshake and authorize every command instead of trusting a connected socket forever.

A viewer may receive public events, while only an on-call operator can acknowledge an incident. That contrast gives us something useful to test instead of a rule to memorize.

Connection authentication establishes identity once; authorization still belongs at every privileged action and can change during a session. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can put a long-lived secret in the WebSocket URL and treat connection success as permanent authorization and still get one successful demo. Push past the demo. use secure handshake credentials, check origin where relevant, authorize each command, and close or refresh stale sessions, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Build a bidirectional operator channel with explicit messages, bounded queues, and observable connection state. Save the before-and-after evidence now; it will make the final project review much more honest.

Test an anonymous viewer, an operator, a revoked operator, and a cross-origin connection against the same channel. Save the evidence, then explain which requirement would force you to choose a different design.

Bound the message flow

Protect both peers from producers that send faster than consumers can process.

The board coalesces viewer-count updates and disconnects a client whose outbound queue exceeds a safe limit. This is a small part of a live service-status board that streams incidents without losing its ordinary HTTP foundation, but the boundary it creates affects everything that follows.

The classic browser WebSocket API queues outgoing data and exposes `bufferedAmount`, but it does not provide automatic backpressure for incoming messages. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to append every update to an unbounded array until memory or responsiveness collapses. It makes later failures harder to locate. Instead, set message and queue limits, coalesce replaceable updates, monitor buffered bytes, and define a slow-client policy. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Build a bidirectional operator channel with explicit messages, bounded queues, and observable connection state. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Throttle one client, send a burst of updates, and prove memory stays bounded and important incident events remain ordered. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: build a websocket channel

Check upgrades, message envelopes, authentication, flow control, heartbeats, and close behavior.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Make live state reliable

Survive ordering, duplicates, gaps, reconnects, restarts, and multiple tabs.

Handle order and duplicates

Use stable event identifiers and idempotent application so retries do not corrupt state.

Before choosing a tool or writing more code, make the requirement concrete. Applying `incident.updated:84` twice should produce the same board, while an older version must not overwrite a newer one.

Networks retry, reconnect, buffer, and sometimes deliver duplicates; application state must not depend on every event arriving exactly once. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to increment local state blindly for every received message. The happy path may still work, which makes the mistake easy to miss. track stable ids or versions, ignore duplicates, reject stale updates, and make commands idempotent where retries are possible. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Make the final state correct through duplicates, gaps, restarts, multiple tabs, and temporary network loss. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Replay the same event twice and then replay an older event; assert that the visible board does not regress. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Repair with snapshots

Treat the event stream as a sequence of changes and the snapshot as the source for rebuilding current state.

Start from the behavior you can observe. After a server restart, the board loads current incidents and continues from the snapshot event id rather than demanding infinite history.

A replay log can be bounded because a complete snapshot gives clients a new known position. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to retain every event forever solely because clients might reconnect. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to define snapshot consistency, include a continuation position, and document when clients must discard local projections. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Make the final state correct through duplicates, gaps, restarts, multiple tabs, and temporary network loss. Record enough evidence that another developer can repeat the result without relying on your memory.

Restart the server, expire the replay window, and recover a stale client using one snapshot plus later events. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Reconnect with backoff

Use jittered exponential backoff and clear connection states so an outage does not become a retry storm.

The status board moves through connecting, live, reconnecting, stale, and offline states instead of showing an eternal green dot. That contrast gives us something useful to test instead of a rule to memorize.

Immediate synchronized reconnects multiply an outage; backoff spreads attempts while the UI explains that data may be stale. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can retry every connection in a tight fixed loop and hide the state from the user and still get one successful demo. Push past the demo. cap a jittered exponential delay, reset it after stability, and render the age of the last confirmed update, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Make the final state correct through duplicates, gaps, restarts, multiple tabs, and temporary network loss. Save the before-and-after evidence now; it will make the final project review much more honest.

Simulate a five-minute outage with many clients and graph reconnect attempts per second. Save the evidence, then explain which requirement would force you to choose a different design.

Coordinate browser tabs

Use BroadcastChannel to share same-origin updates or leadership without opening redundant live connections.

One elected tab can own the server connection and relay updates, while every tab can fall back if the leader disappears. This is a small part of a live service-status board that streams incidents without losing its ordinary HTTP foundation, but the boundary it creates affects everything that follows.

Same-origin browsing contexts can exchange messages through BroadcastChannel, but the channel is local to the browser and is not durable storage. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to assume BroadcastChannel reaches other devices or preserves events after every tab closes. It makes later failures harder to locate. Instead, define a same-origin tab protocol with leader expiry, handover, duplicate tolerance, and an independent snapshot recovery path. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Make the final state correct through duplicates, gaps, restarts, multiple

tabs, and temporary network loss. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Open three tabs, close the leader, and verify another tab reconnects without losing the current incident state. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: make live state reliable

Check ordering, duplicates, snapshots, reconnect backoff, cross-tab coordination, and presence semantics.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Secure, test, and operate

Threat-model, load-test, observe, and degrade the live feature safely.

Threat-model the live channel

Review origins, credentials, message validation, authorization, rate limits, and data exposure together.

Before choosing a tool or writing more code, make the requirement concrete. The public incident feed exposes only public fields, while operator messages are authenticated, authorized, rate-limited, and audited.

A long-lived connection expands the time and volume over which an attacker can probe inputs and consume resources. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to focus on transport encryption and ignore malicious messages or expensive connection behavior. The happy path may still work, which makes the mistake easy to miss. map trust boundaries, minimize payloads, validate every message, limit connections and rates, and log privileged actions. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Ship a live feature that has limits, useful telemetry, a degradation path, and a repeatable failure test matrix. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Write abuse cases for connection floods, oversized messages, forged commands, stale credentials, and sensitive event fields. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Load-test connections and fanout

Measure concurrent connections, event fanout, queue growth, CPU, memory, and reconnect storms.

Start from the behavior you can observe. One incident sent to ten thousand quiet viewers creates different pressure from ten thousand operators sending commands.

Live systems often fail from connection count and fanout shape rather than ordinary request throughput. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to benchmark a single fast local client and extrapolate from it. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to test representative connection lifetimes, message sizes, fanout, slow clients, and synchronized recovery events. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Ship a live feature that has limits, useful telemetry, a degradation path, and a repeatable failure test matrix. Record enough evidence that another developer can repeat the result without relying on your memory.

Run a staged load test and record the first resource that becomes limiting rather than chasing a vanity connection number. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Observe the connection lifecycle

Instrument opens, closes, reasons, reconnects, lag, queue depth, and last applied event position.

A dashboard shows current connections, abnormal close codes, replay gaps, slow consumers, and the age of the newest delivered event. That contrast gives us something useful to test instead of a rule to memorize.

Request counts alone cannot explain a live system; you need connection and stream lifecycle signals. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can log every payload while omitting the metrics needed to spot a system-wide failure and still get one successful demo. Push past the demo. collect low-cardinality lifecycle metrics, structured errors, traceable ids, and redacted samples where they add evidence, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Ship a live feature that has limits, useful telemetry, a degradation path, and a repeatable failure test matrix. Save the before-and-after evidence now; it will make the final project review much more honest.

Create an alert for stale delivery and use the metrics to distinguish server silence from client reconnect failure. Save the evidence, then explain which requirement would force you to choose a different design.

Degrade gracefully and finish the board

Complete the project with a stale-state warning, manual refresh, snapshot fallback, and documented limits.

If streaming is unavailable, the board keeps its last confirmed data, labels its age, polls cautiously, and offers a refresh. This is a small part of a live service-status board that streams incidents without losing its ordinary HTTP foundation, but the boundary it creates affects everything that follows.

A live feature should become less live during failure, not become unusable or silently wrong. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to remove the ordinary HTTP path after the live demo succeeds. It makes later failures harder to locate. Instead, keep the snapshot route, expose freshness, switch to a bounded fallback, and test recovery back to the preferred channel. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Ship a live feature that has limits, useful telemetry, a degradation path, and a repeatable failure test matrix. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Run the complete matrix: first load, duplicate event, stale event, disconnect, server restart, slow client, revoked user, and fallback. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: secure, test, and operate

Check threat boundaries, load tests, observability, deployment limits, degradation, and final verification.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/real-time-web-applications/.