Asynchronous

Arnav Gupta

April 18, 2024

Contents

1 UI Responsiveness

Responsive UIs deliver feedback to the user in a timely manner. User does not wait longer than necessary and state of long tasks is communicated to the user.

Can make a UI responsive by:

1. designing to meet human expectations and perceptions
2. loading data efficiently so it is quickly available

Responsiveness is affected by:

- user exectations: how quickly a system should react to complete some task
  - expectations for technology (ex. web vs desktop)
- application and interface design
  - interface keeps up with user actions
  - interface informs user about application status
  - interface doesn't make user wait unexpectedly

Responsiveness is the most important factor is the most important factor in user satisfaction, more than ease of learning/use.

Responsiveness is **not** just system performance.

If slow performance, to keep the UI responsive:

- provide feedback to confirm user actions (let user know input received)
- provide feedback about events (how long an operation will take)
- allow users to perform other tasks while waiting
- anticipate the user's most common requests (pre-fetch data below current scroll vew)
- perform low-priority system tasks in the background

Knowing duration of perceptual and cognitive processes can inform design of interactive systems that feel responsive:

- minimal time to detect gap of silence in sound: 4 ms
- minimal time to be affected by visual stimulus: 10 ms
  - continuous input latency should be less than this
- time that vision is supressed during saccade: 100 ms
- max interval between cause-effect events: 140 ms
  - if UI feedback takes longer, perception of cause-effect is broken
- time to comprehend printed word: 150 ms
- visual motor reaction time to observed event: 1 s
  - display busy/progress indicators for operations >1 s
  - present a skeleton screen
- time to prepare for conscious cognition task: 10 s
  - display a fake version of application interface or image of document on last save while real one loads in < 10s
- duration of unbroken attention to a single task: 6 s to 30 s

Skelton screens are a minimal version of an interface while the real one loads (generic layout or minimal version of content).

- user adjusts to layout
- loading process seems faster since there is an initial result

Progress indicator best practices:

- show work remaining, not completed
- show total progress when multiple steps, not only step progress
- display finished state briefly at the end
- show smooth progress, not bursts
- use human precision, not computer precision

Progressive loading involves providing user with some data while loading the rest of the data.

Predicting next operation involved using periods of low load to pre-compute resources to high probability requests, which speeds up subsequent responses.

Graceful degradation of feedback involves simplifying feedback for high-computation tasks.

Goals for handling long tasks in a UI are:

- to keep the UI responsive
- provide progress feedback
- ideally allow long task to be paused or cancelled

2 Worker Thread

With multi-threading, manage multiple concurrent threads with shared resources, but executing different instructions.

Threads divide computation and reduce blocking. Concurrency risks exist, such as 2 threads updating a variable.

Browsers support **worker threads**: dedicated works, shared works, and service workers.

3 Fetch API

The JS runtime environment has a JS Engine, Web API, and message queue.

JS Engine has an execution context stack (ECS), with code to execute next, and a heap, with function definitions, objects, etc.

Web API has a DOM API for handling DOM events, Fetch API for loading remote resources (cloud API, server data, etc), and timer functions (`setTimeout`, `setInterval`, `requestAnimationFrame`).

Message queue holds methods for events that have occurred. Methods that generate messages are stored in the event table.

To execute an asynchronous method from the message queue, it must be moved to the ECS. An event loop continually checks if the ECS is empty, and if it is, it moves a method from the message queue to the ECS to the JS Engine can execute it.

Input events are asynchronous methods (callbacks bound to a DOM element).

Fetch API is an interface for fetching resources across the network. The `fetch` function starts the process of fetching a resource from the network and returns a `Promise` object with 3 possible states:

- *pending*: when fetch process is happening
- *resolved*: when process was successful and has a valid response
- *rejected*: when the process failed and has an error

Complex to get progress during fetch.