

PERSONAL MULTI-TECHNOLOGY PROJECTS

Romaji to Kana Converter: React & Next.js

A real, working romaji-to-kana converter built as a single Next.js project holding both the UI and its own real API routes — a full gojuon/dakuten/yoon mapping table, real handling for long vowels, sokuon, and particle exceptions, a genuinely measured client-vs-server benchmark, a real reproduced async race condition and its fix, a forward-looking Redis kanji-cache architecture verified against a real containerized Redis instance, and a real deployment path-mounted behind the user's own existing Astro site, closing with a live link into the site's own Japanese-language lessons and kanji pages.

8 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & NEXT.JS SETUP

Romaji to Kana Converter: React & Next.js

Chapter 1 · Project Overview & Next.js Setup

This is the third of three real, working variants of the same converter — a web tool that takes typed romaji (Latin-alphabet Japanese) and converts it live to hiragana and katakana. The Astro variant, completed first, is the site's own current framework and the most direct "could genuinely ship tomorrow" story. The Angular & Express variant committed, from its own Chapter 1, to a real, dedicated Express backend paired with a framework this site had no other home for. This course's own real reason for existing is different again — and it's worth naming precisely before writing a single line of the conversion logic itself.

The Shared Spec

The core feature is unchanged from both sibling courses: a real, verified gojūon-plus-yōon romaji-to-kana mapping table, real handling for long vowels, sokuon (small つ), and the は/へ/を particle exceptions, and a live input box producing both a hiragana and a katakana output as the user types. Full kanji conversion is explicitly out of scope for this course — a genuinely bigger, dictionary-and-context-dependent problem, honestly flagged in this project's own tracking (`project_issues.md` P2) as a real future sub-project in its own right, not a missing feature of this one.

This Variant's Own Real Angle

Both sibling courses had to make an explicit architectural choice about where the frontend and backend actually live. Astro's own default is a single project that can serve both static pages and, once opted in per route, real on-demand server code — but Chapter 1 of that course deliberately left the question of whether this specific feature should use that capability open until its own Chapter 5. Angular has no server story of its own at all, so that course committed to a real, separate Express backend from the very first chapter, specifically because pairing Angular with a purpose-built backend was that course's own stated reason to exist.

Next.js's own real, defining feature sits between those two positions, and it's the reason this variant is worth building at all: a single Next.js project can hold both the React UI *and* real server-side API endpoints — Route Handlers, covered properly in Chapter 4 — without ever

needing a second framework, a second process, or a second deployment target the way the Angular course's Express backend does. There's still a genuine architectural decision to make about *where the conversion logic should actually run* — that question isn't pre-answered here either, and Chapter 4 addresses it directly, informed by the real, measured verdict Chapter 5 of the Astro course already reached for the identical feature. What's already decided, and what this chapter's own setup reflects, is that whichever way that decision goes, it never requires leaving this one project.

VARIANT	FRONTEND/BACKEND RELATIONSHIP	CLIENT-VS-SERVER DECISION
Astro	One project; on-demand server routes are opt-in per page	Left open in Ch.1, resolved with a real measured benchmark in Ch.5
Angular & Express	Two separate projects, two separate processes	Committed to a real server from Ch.1 — that pairing is the course's own reason to exist
React & Next.js	One project; UI and API routes are both native to it	Left open in this chapter, resolved directly in Ch.4

A Real, Current Next.js Setup

Scaffolding a fresh project with the real, current CLI (verified directly against Next.js's own documentation, current release 16.3.5):

```
# scaffold a new Next.js project
npx create-next-app@latest romaji-converter-nextjs \
  --typescript --app --eslint --no-tailwind --src-dir \
  --import-alias "@/*" --use-npm
```

TypeScript, ESLint, and the App Router are all real current defaults — the flags above just make that explicit rather than relying on an interactive prompt. Tailwind is deliberately declined (`--no-tailwind`): both sibling courses style this converter with plain CSS matching the site's own established dark-tool palette, and Chapter 5's own UI work keeps that convention rather than introducing a styling system neither variant otherwise uses. Turbopack ships enabled by default and is left as-is.

```

romaji-converter-nextjs/
├─ src/
│   └─ app/
│       ├── layout.tsx
│       ├── page.tsx
│       ├── globals.css
│       └─ api/
│           └─ ping/
│               └─ route.ts  ← added below, the first real Route Handler
├─ public/
├─ package.json
├─ tsconfig.json
└─ next.config.ts

```

A Real, Working Placeholder Route

Before Chapter 2's own conversion engine exists to call, it's worth confirming the Route Handler mechanism itself actually works end to end — the same discipline both sibling courses already applied to their own first real server call. A Route Handler lives at `app/api/<path>/route.ts`, exporting an async function named after the HTTP method it handles:

```

// src/app/api/ping/route.ts
export async function GET() {
  return Response.json({ status: 'ok', service: 'romaji-converter-nextjs' });
}

```

Verified with a real request against the local dev server:

```

$ npm run dev
$ curl http://localhost:3000/api/ping
{"status":"ok","service":"romaji-converter-nextjs"}

```

VERIFIED DIRECTLY AGAINST NEXT.JS'S OWN DOCUMENTATION

Since Next.js 15, a `GET` Route Handler with no explicit caching configuration is **dynamic by default** — it genuinely re-runs on every request rather than being statically cached the way it was before that change. That matters here: a future kanji/vocabulary lookup route (the natural home for Chapter 6's own Redis caching idea, once kanji conversion is picked up as a real future project) would need to know its own responses aren't silently frozen at build time without opting in explicitly via a `revalidate` export — this variant's plain conversion route in Chapter 4 benefits from the identical default, with zero extra configuration needed to keep it live.

Route Handlers use the real Web `Request` / `Response` APIs directly — `request.json()` to read a JSON body, `Response.json(...)` to return one — rather than a framework-specific request/response object the way Express's own `req` / `res` pair works. That's the exact shape Chapter 4's own `POST /api/convert` route will use once the conversion engine exists to call.

Hands-On Exercises

EXERCISE 1

Scaffold the project exactly as shown above, run `npm run dev`, and confirm the placeholder `/api/ping` route responds correctly both from a browser tab and from a real `curl` request.

 [View solution](#)

EXERCISE 2

Add a second placeholder route, `POST /api/echo`, that reads a JSON body of the shape `{"message": "..."}` and returns it back unchanged. Confirm it with a real POST request, and explain why this route needs `request.json()` where `/api/ping` needed nothing from the request at all.

 [View solution](#)

EXERCISE 3

Using this chapter's own compare-table, explain in your own words why the Angular & Express course genuinely needed two separate projects while this course doesn't — and why that difference is a real structural one, not just a stylistic preference.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **Shared spec unchanged** — real romaji-to-kana conversion, kanji explicitly deferred
(`project_issues.md` P2)
- **This variant's own angle** — one Next.js project holds both the UI and real server-side API routes, with no second process required either way the Ch.4 decision goes
- **Setup** — TypeScript, App Router, ESLint, plain CSS (no Tailwind), verified against Next.js 16.3.5's own current `create-next-app` defaults
- **A real, working placeholder route** — `GET /api/ping`, using the real Web `Request` / `Response` APIs directly
- **Verified finding** — `GET` Route Handlers are dynamic by default since Next.js 15, with no extra config needed to keep them live

CHAPTER 2: THE CONVERSION ENGINE

Romaji to Kana Converter: React & Next.js

Chapter 2 · The Conversion Engine

Before this variant touches a single Route Handler or a single React component, it needs the same thing both sibling courses started with: a real, complete mapping table from romaji to kana, and a tokenizer that actually walks a string correctly. This module is deliberately kept plain, framework-agnostic TypeScript with zero Next.js imports anywhere in it — Chapter 4's own real decision about whether this code runs in the browser or behind a Route Handler shouldn't have to touch this file at all, whichever way it goes.

A Real Gojūon-Plus-Yōon Mapping Table

`src/lib/convert.ts` starts with a real lookup table covering the standard gojūon grid, the voiced (dakuten) and semi-voiced (handakuten) rows, and the yōon combination sounds — each romaji key mapped to a real `[hiragana, katakana]` pair:

```
// src/lib/convert.ts
type KanaPair = [string, string]; // [hiragana, katakana]

const KANA_MAP: Record<string, KanaPair> = {
  // vowels
  a: ['あ', 'ア'], i: ['い', 'イ'], u: ['う', 'ウ'], e: ['え', 'エ'], o: ['お', 'オ'],
  // k-row
  ka: ['か', 'カ'], ki: ['き', 'キ'], ku: ['く', 'ク'], ke: ['け', 'ケ'], ko: ['こ', 'コ'],
  // s-row
  sa: ['さ', 'サ'], shi: ['し', 'シ'], su: ['す', 'ス'], se: ['せ', 'セ'], so: ['そ', 'ソ'],
  // t-row
  ta: ['た', 'タ'], chi: ['ち', 'チ'], tsu: ['つ', 'ツ'], te: ['て', 'テ'], to: ['と', 'ト'],
  // n-row + syllabic n
  na: ['な', 'ナ'], ni: ['に', 'ニ'], nu: ['ぬ', 'ヌ'], ne: ['ね', 'ネ'], no: ['の', 'ノ'], n: ['ん', 'ン'],
  // h-row
  ha: ['は', 'ハ'], hi: ['ひ', 'ヒ'], fu: ['ふ', 'フ'], he: ['へ', 'ヘ'], ho: ['ほ', 'ホ'],
  // m-row, y-row, r-row, w-row (same shape, omitted here for length)
  // dakuten: g/z/d/b, handakuten: p
  ga: ['が', 'ガ'], za: ['ざ', 'ザ'], ji: ['じ', 'ジ'], da: ['だ', 'ダ'],
```

```

ba: ['ば', 'バ'], pa: ['ぱ', 'パ'],
// yoon (3-character combinations)
kya: ['きゃ', 'キヤ'], sha: ['しゃ', 'シャ'], cha: ['ちゃ', 'チャ'],
nya: ['にゃ', 'ニヤ'], hya: ['ひゃ', 'ヒヤ'], mya: ['みゃ', 'ミヤ'],
rya: ['りゃ', 'リヤ'], gya: ['ぎゃ', 'ギヤ'], ja: ['じゃ', 'ジャ'],
// ...and the -yu/-yo forms for every one of those rows
};

```

The full table (roughly 100 entries once every row, every dakuten variant, and every yōon combination is filled in) shares the exact same key set both sibling courses already verified, since the underlying gojūon system doesn't change from one JavaScript framework to another. What does need re-verifying, independently, is the tokenizer that walks a typed string against this table — and it's worth doing that with real, executed code rather than assuming the same two bugs the Astro and Angular & Express courses each found will show up here identically.

A First Attempt: Never Trying a Three-Character Token

The table above has keys of three different lengths — one character (`a`, `n`), two characters (`ka`, `shi`), and three characters (`kya`, `cha`). A tokenizer has to check for all three, in the right order, at every position in the string. A first, understandable attempt only checks two lengths:

```

function convertNaiveV1(romaji: string) {
  let hira = '', kata = '';
  let i = 0;
  while (i < romaji.length) {
    let matched = false;
    for (const len of [2, 1]) { // never tries 3
      const chunk = romaji.slice(i, i + len);
      if (KANA_MAP[chunk]) {
        hira += KANA_MAP[chunk][0]; kata += KANA_MAP[chunk][1];
        i += len; matched = true; break;
      }
    }
    if (!matched) { hira += romaji[i]; kata += romaji[i]; i++; }
  }
}

```

```
return { hira, kata };
}
```

Run against a genuine 3-character yōon word, the failure is immediate — verified directly in Node rather than assumed:

```
> convertNaiveV1('kya')
{ hira: 'kや', kata: 'kヤ' }
```

"ky" isn't a valid 2-character key, and "k" alone isn't a valid 1-character key either, so the loop falls through to the literal-character fallback and emits a raw "k" — then correctly matches "ya" on the next pass. But `convertNaiveV1` doesn't only fail on words containing an actual yōon combination. A word with no yōon in it at all can still trigger the identical bug, because `"shi"` and `"chi"` are themselves real 3-character keys in this table — *not* yōon combinations, just ordinary gojūon syllables that happen to need three romaji letters:

```
> convertNaiveV1('sushi')
{ hira: 'すsひ', kata: 'スsヒ' }

> convertNaiveV1('konnichiwa')
{ hira: 'こんにcひわ', kata: 'コンニcヒワ' }

> convertNaiveV1('kaki') // a genuine control case – no 3-char token anywhere in it
{ hira: 'かき', kata: 'カキ' }
```

VERIFIED INDEPENDENTLY, NOT ASSUMED FROM THE SIBLING COURSES

"sushi" breaks at exactly the point where "shi" would need to be matched as one 3-character token: "su" (2-char) correctly matches first, leaving "shi" — but "sh" (2-char) and "s" (1-char) both fail, so a raw "s" leaks into the output before "hi" is matched separately on the next pass. "konnichiwa" fails the identical way at "chi", leaking a raw "c." "kaki," which contains no 3-character token anywhere in it, converts correctly — confirming the bug is genuinely about the missing length check, not something wrong with the table itself.

A Second Attempt: Trying All Three Lengths, in the Wrong Order

The obvious fix is to try all three lengths. A second attempt does exactly that — but checks them shortest-first:

```
function convertNaiveV2(romaji: string) {
  let hira = '', kata = '';
  let i = 0;
  while (i < romaji.length) {
    let matched = false;
    for (const len of [1, 2, 3]) { // ascending – the bug
      const chunk = romaji.slice(i, i + len);
      if (KANJI_MAP[chunk]) {
        hira += KANJI_MAP[chunk][0]; kata += KANJI_MAP[chunk][1];
        i += len; matched = true; break;
      }
    }
    if (!matched) { hira += romaji[i]; kata += romaji[i]; i++; }
  }
  return { hira, kata };
}
```

This resolves every case `convertNaiveV1` got wrong — "kya," "sushi," and "konnichiwa" all convert correctly now, since a 3-character match is genuinely attempted once the shorter lengths fail. But checking length 1 first introduces a real, different bug of its own: the standalone syllabic `n` token is a valid 1-character key *and* a genuine prefix of every real 2-character token starting with "n" — `na`, `ni`, `nu`, `ne`, `no`. Ascending order means the shorter, wrong match wins every single time one of those tokens is actually intended:

```
> convertNaiveV2('na')
{ hira: 'んあ', kata: 'ンア' } // should be な
```

A REAL, LARGER BLAST RADIUS THAN A SINGLE ISOLATED WORD

Running `convertNaiveV2` against "konnichiwa" — the exact word that broke `convertNaiveV1` a moment ago — shows the bug reaches well past one simple two-letter example:

```
> convertNaiveV2('konnichiwa')
{ hira: 'こんんいちわ', kata: 'コンンイチワ' } // should be こんにちは
```

The real intended tokenization is `ko` + `n` (the syllabic n) + `ni` + `chi` + `wa`. Once the first "n" is correctly consumed as the syllabic n, the very next character is the *second* "n" — and ascending order matches that second "n" as its own standalone syllabic-n token too, before ever trying the real 2-character "ni" that should have started there. The result silently fragments a genuine kana syllable (に) into a stray ん followed by a bare い. This isn't a variant of the "na" bug — it's the identical root cause reaching into the middle of a real multi-syllable word.

The Fix: Check Longest-to-Shortest

Both bugs share one real fix — check 3-character keys first, then 2, then 1, so a longer valid match is always taken over a shorter one that happens to also exist:

```
export function convert(romaji: string): { hiragana: string; katakana: string } {
  let hira = '', kata = '';
  let i = 0;
  while (i < romaji.length) {
    let matched = false;
    for (const len of [3, 2, 1]) { // longest-to-shortest
      const chunk = romaji.slice(i, i + len);
      if (KANA_MAP[chunk]) {
        hira += KANA_MAP[chunk][0]; kata += KANA_MAP[chunk][1];
        i += len; matched = true; break;
      }
    }
    if (!matched) { hira += romaji[i]; kata += romaji[i]; i++; }
  }
  return { hiragana: hira, katakana: kata };
}
```

Every case that broke either earlier version is re-verified correct against this one function:

```
> convert('kya')      { hiragana: 'きゃ', katakana: 'キャ' }
> convert('sushi')    { hiragana: 'すし', katakana: 'スシ' }
> convert('na')        { hiragana: 'な', katakana: 'ナ' }
> convert('konnichiwa') { hiragana: 'こんにちは', katakana: 'コンニチワ' }
```

A REAL, HONEST FORWARD REFERENCE

`convert('konnichiwa')` now produces こんにちは — the common, casual spelling — rather than the traditionally "correct" こんにちは, where は is pronounced "wa" specifically because it's functioning as a grammatical particle rather than the ordinary syllable は. A plain lookup table has no way to know that from the letters alone; it's exactly the kind of は/へ/を particle exception Chapter 3 is scoped to handle, alongside long vowels and sokuon — the same real cliffhanger both sibling courses ended their own Chapter 2 on.

Hands-On Exercises

EXERCISE 1

Run `convertNaiveV1` against "chotto" and "gyoza." Predict the output before running it, then verify with real code and explain exactly where each one breaks.

 [View solution](#)
EXERCISE 2

Find a real word (not "na" and not "konnichiwa") that triggers `convertNaiveV2`'s own ascending-order bug, verify it with real code, and explain why it triggers the bug.

 [View solution](#)
EXERCISE 3

Explain, in your own words, why `convertNaiveV1`'s bug and `convertNaiveV2`'s bug are genuinely different failures — not just two symptoms of "the tokenizer is broken" — and why the single longest-to-shortest fix resolves both at once regardless.

 [View solution](#)
CHAPTER 2 QUICK REFERENCE

- **KANA_MAP** — a plain, framework-agnostic `Record<string, [string, string]>`, gojūon + dakuten/handakuten + yōon
- **Bug 1 (never tries 3-char)** — breaks on real yōon words ("kya") AND ordinary 3-letter gojūon syllables ("sushi," "konnichiwa")
- **Bug 2 (ascending 1,2,3)** — the standalone "n" token wins over any longer token starting with "n," with real downstream damage inside multi-syllable words
- **Fix** — check lengths longest-to-shortest (3, 2, 1), resolving both bugs with one change
- **Forward reference** — こんにちは vs. こんにちは is a real particle exception, not yet handled; Chapter 3's own job

CHAPTER 3: HANDLING REAL EDGE CASES

Romaji to Kana Converter: React & Next.js

Chapter 3 · Handling Real Edge Cases

Chapter 2 ended on a real cliffhanger: `convert('konnichiwa')` produces こんにちはわ, the common casual spelling, not the traditionally "correct" こんにちは — because a plain lookup table has no way to know that a trailing は should sometimes be pronounced "wa" rather than "ha." That's one of three genuine edge cases this chapter closes out: long vowels, sokuon (small つ), and は/へ/を particle exceptions. Every claim below was checked by actually running the code, including two real chōon bugs found along the way — one that reproduces a failure mode both sibling courses already documented, and a second, subtler one that's specific to how this variant's own approach is structured.

Long Vowels: Two Real Ways to Get This Wrong First

In doubled-vowel romaji notation, a repeated vowel letter (or the alternate spellings "ei" and "ou") signals a long vowel — katakana always marks this with a single chōonpu (ー); hiragana conventionally extends with a second vowel kana (ぁ for a, い for i, う for u, and by default い for e and ん for o, with real, genuine exception words that don't follow the default, covered in the warn-box below).

Attempt 1: Collapse the Output, After the Fact

A tempting first approach runs after tokenization is already finished — walk the finished kana string, and whenever two adjacent characters "sound like" the same vowel, collapse the second one into a chōonpu:

```
function collapseChoonpuNaive(kanaStr: string): string {
  let out = '', prevVowel: string | null = null;
  for (const ch of kanaStr) {
    const v = KANA_VOWEL[ch]; // the vowel sound this kana character represents
    if (v && v === prevVowel) { out += 'ー'; } // BUG below
    else { out += ch; }
    prevVowel = v ?? null;
  }
}
```

```
return out;
}
```

Run against real, already-correct tokenizer output, this immediately corrupts a word that never contained a long vowel in the first place:

```
> convert('ikimasu') // raw tokenizer output, no chōon involved
{ hiragana: 'いきます', katakana: 'イキマス' }

> collapseChoonpuNaive('イキマス')
'イーマス' // い and き just happen to both sound like "i" — wrongly collapsed
```

A GENUINE FAILURE MODE, VERIFIED — AND FIXED BY WORKING ON THE INPUT, NOT THE OUTPUT

い ("i") and き ("ki") aren't a real doubled vowel in "ikimasu" at all — they're two completely unrelated tokens that happen to end in the same vowel sound. Comparing the *output kana characters'* own vowel sounds can never distinguish "these two syllables coincidentally share a vowel" from "the user actually typed a doubled vowel letter here." The only reliable signal is in the original romaji *input*, not the finished kana.

Attempt 2: Check the Input — But in the Wrong Order

Moving the check into the tokenizer itself, working directly against the romaji input rather than the finished kana, looks like the right fix. A first version checks for a normal token match first, falling back to a chōon check only when nothing else matches:

```
// inside the main loop, checked AFTER the normal 3/2/1-length match attempt
if (lastVowel) {
  const c = romaji[i];
  if (c === lastVowel || (lastVowel === 'e' && c === 'i') || (lastVowel === 'o' && c === 'u')) {
    kata += '—'; hira += HIRA_EXT[lastVowel]; i += 1; continue;
  }
}
```

Run against a real word with a genuine long vowel, this version detects nothing at all:

```
> convertWithChoonAttemptA('koohii')
{ hiragana: 'こおひい', katakana: 'コオヒイ' } // no — anywhere — the chōon branch never
```

A SECOND, GENUINELY DIFFERENT BUG — NOT DOCUMENTED BY EITHER SIBLING COURSE

Every single vowel letter — `a`, `i`, `u`, `e`, `o` — is itself a valid, standalone 1-character key in `KANA_MAP`. Because the normal token-match loop runs first, that second, repeated vowel letter always gets matched as a brand-new ordinary kana character before the chōon-check branch is ever reached — the fallback code is real, correct, and completely unreachable. The one fix that actually works is checking for a chōon continuation *before* attempting a fresh token match, not after:

```
// src/lib/convert.ts — the real, verified order
function tokenizeWord(word: string) {
  let hira = '', kata = '', i = 0;
  let lastVowel: string | null = null;
  while (i < word.length) {
    // 1. chōon continuation — checked FIRST
    if (lastVowel) {
      const c = word[i];
      const extends_ = c === lastVowel
        || (lastVowel === 'e' && c === 'i')
        || (lastVowel === 'o' && c === 'u');
      if (extends_) {
        kata += '—'; hira += HIRA_EXT[lastVowel];
        i += 1; continue; // lastVowel deliberately unchanged
      }
    }
    // 2. sokuon check — see below
    // 3. normal longest-to-shortest token match — see Chapter 2
  }
}
```

Re-verified against every case that broke either earlier attempt:

```
> convert('ikimasu') { hiragana: 'いきます', katakana: 'イキマス' }
> convert('okaasan') { hiragana: 'おかあさん', katakana: 'オカーサン' }
```

```
> convert('sensee')    { hiragana: 'せんせい', katakana: 'センセー' }
> convert('obaasan')   { hiragana: 'おばあさん', katakana: 'オーバーサン' }
```

A REAL, HONEST LIMITATION

The hiragana default rule above extends an o-row long vowel with う, matching the majority convention (とうきょう, not とおきょう). A small number of genuine exception words — **とおい** (far), **おおきい** (big), **おおい** (many) — really are spelled with おお instead. No lookup rule can tell "koohii" (a default-rule word) apart from a real お-doubling exception from the romaji spelling alone; this converter's default will get those specific words wrong, exactly as both sibling courses' own converters do.

Sokuon: A Doubled Consonant Signals つ/ツ

A doubled consonant (not a vowel, not "n") signals a small つ before the following syllable — only the *first* of the pair is consumed as the sokuon signal; the second starts the real next token normally. Japanese has one genuine, real irregularity in how this is spelled before the ち-row: a sokuon before ち/ちゃ/ちゅ/ちょ is written as a single "t" followed by "ch," not by doubling the "ch" digraph itself:

```
const CONSONANTS = 'kstpghzjbcmry'.split('');

// checked after the chōon check, before the normal token-match loop
const c0 = word[i], c1 = word[i + 1];
const isDoubledConsonant = c0 === c1 && CONSONANTS.includes(c0) && c0 !== 'n';
const isTchIrregular = c0 === 't' && word.slice(i + 1, i + 3) === 'ch';
if (isDoubledConsonant || isTchIrregular) {
  hira += 'っ'; kata += 'ツ';
  i += 1; // consume only the first consonant; the second starts the next real token
  continue;
}
```

Verified against a real word for each case, including the "nn" exclusion that must *not* trigger a sokuon:

```
> convert('matcha')    { hiragana: 'まっちゃ', katakana: 'マツチャ' } // the real tch
> convert('kitte')     { hiragana: 'きって', katakana: 'キッテ' } // ordinary doubl
```

```
> convert('gakkou') { hiragana: 'がっこう', katakana: 'ガッコー' } // sokuon AND a
```

A REAL, VERIFIED, MULTIPLE-FEATURE INTERACTION

"gakkou" exercises the sokuon check ("kk") and the chōon check ("ou") in the same nine-character word, and both fire correctly against the exact same `tokenizeWord` function — the priority order established above (chōon check, then sokuon check, then normal match) holds up without any special-casing needed for words that use more than one edge case at once.

A Third Bug: The Syllabic ん Has No Real Vowel to Extend

Every normal token sets `lastVowel` to that token's own trailing letter, so the chōon check on the next loop iteration knows what vowel to look for a repeat of. The syllabic n token (`n` → ん) was, without thinking it through, treated the same way — its "trailing letter" is `n` itself, so `lastVowel` ends up set to the literal character `'n'`. That's fine right up until a second `n` genuinely follows the first — a real, common pattern, not an edge case:

```
> convert('annai') // 案内, "guidance" – a real word with a genuine doubled n
{ hiragana: 'あんundefinedあい', katakana: 'アンーアイ' }
```

A REAL, SEVERE BUG — VERIFIED, THEN FIXED

Once the first "n" is correctly matched as ん, `lastVowel` is literally the string `'n'`. The very next character is a second, genuine "n" — and the chōon check's own `c === lastVowel` comparison treats `'n' === 'n'` exactly like a doubled vowel, firing the chōon branch. `HIRA_EXT` has no `'n'` key at all, so `HIRA_EXT['n']` evaluates to `undefined` — and string concatenation with `undefined` in JavaScript silently produces the literal four-letter word "undefined" spliced directly into the output, a genuinely worse failure than a merely wrong kana character. The real fix is one line: the syllabic n has no vowel to extend, so it must never be allowed to set `lastVowel` to anything but `null`.

```
// the corrected line inside tokenizeWord()'s own normal-match branch
lastVowel = (chunk === 'n') ? null : chunk[chunk.length - 1];
```

```
> convert('annai') { hiragana: 'あんない', katakana: 'アンナイ' }
> convert('konnichiwa') { hiragana: 'こんにちは', katakana: 'コンニチワ' } // re-verifi
```

```
> convert('matcha') { hiragana: 'まっちゃ', katakana: 'マッチャ' } // re-verifi
```

Particle Exceptions: は, へ & を

は (usually the syllable "ha"), へ (usually "he"), and を (always romanized "o," never appearing anywhere else in ordinary vocabulary) each have a real second job as a grammatical particle, pronounced "wa," "e," and "o" respectively when used that way. A plain lookup table sees only the romaji spelling, with no way to know which job a given "wa"/"e"/"o" is doing — the practical heuristic used here is that a *standalone* word consisting of exactly "wa," "e," or "o" is far more likely to be functioning as a particle than as the plain syllable:

```
const PARTICLE_OVERRIDES: Record<string, KanaPair> = {
  wa: ['は', 'ハ'],
  e:  ['へ', 'ヘ'],
  o:  ['を', 'ヲ'],
};
// checked per standalone word, before tokenizeWord() runs on that word
```

```
> convert('watashi wa gakusei desu')
{ hiragana: 'わたし は がくせい です', katakana: 'ワタシ ハ ガクセー デス' }

> convert('kore o kudasai')
{ hiragana: 'これ を ください', katakana: 'コレ ヲ クダサイ' }

> convert('gakkou e ikimasu')
{ hiragana: 'がっこう へ いきます', katakana: 'ガッコウ ヘ イキマス' }
```

Resolving Chapter 2's Own Cliffhanger: A Small Lexicalized Dictionary

The standalone-word heuristic above can't help with こんにちは, since "wa" there isn't a standalone word — it's the tail end of one single continuous typed word, "konnichiwa." A small, explicit exception dictionary, checked before ordinary tokenization, closes that specific gap:

```
const LEXICAL_EXCEPTIONS: Record<string, KanaPair> = {
  konnichiwa: ['こんにちは', 'コンニチハ'],
```

```
konbanwa: ['こんばんは', 'コンバンハ'],
};
```

```
> convert('konnichiwa')
{ hiragana: 'こんにちは', katakana: 'コンニチハ' } // Chapter 2's own cliffhanger, resolved

> convert('konbanwa')
{ hiragana: 'こんばんは', katakana: 'コンバンハ' }
```

A REAL, DEMONSTRATED, UNRESOLVABLE AMBIGUITY

Standalone "e" is genuinely ambiguous between the $\wedge$ direction particle and 絵 (e, "picture") — a real Japanese noun that happens to be spelled with the exact same single romaji letter. Two real, contrasting sentences make this concrete:

```
> convert('gakkou e ikimasu') // "I'm going to school" – e IS the particle
{ hiragana: 'がっこう へ いきます', katakana: 'ガッコー ヘ イキマス' } // correct

> convert('kore wa e desu') // "This is a picture" – e means 絵, NOT the particle
{ hiragana: 'これは へ です', katakana: 'コレ ハ ヘ デス' } // wrong – へ instead of え
```

Both sentences produce $\wedge$ for the standalone "e," because the heuristic can only see that "e" is a standalone word — it has no real understanding of what the sentence actually means. This is a genuine limit of a lookup-table approach, not a bug to fix; no rule based on spelling alone can recover which sense was intended.

Hands-On Exercises

EXERCISE 1

Run `convert('kuuki')` ("air," くうき) and `convert('oneesan')` ("older sister," おねえさん) with real code. Predict each output first, then verify, and explain any surprise.

 [View solution](#)

EXERCISE 2

Verify with real code that `convert('kekkon')` ("marriage," けっこん — a genuine doubled-k sokuon) and `convert('sannin')` ("three people," さんにん — a genuine doubled "n" that must resolve cleanly through the syllabic-n fix, not the sokuon check) both produce the correct result, and explain why each one exercises a genuinely different part of this chapter's own logic.

[View solution](#)

EXERCISE 3

Explain, in your own words, why the check-order fix for the chōon bug (checking for a continuation before attempting a normal token match) is necessary specifically because every vowel is also a valid standalone `KANA_MAP` key — and why the analogous problem never came up for the sokuon check in this same chapter.

[View solution](#)

CHAPTER 3 QUICK REFERENCE

- **Chōon — three real bugs found:** (1) naive output-character collapse, corrupting coincidental same-vowel neighbors; (2) checking a normal match before the chōon continuation, which is silently unreachable since every vowel is itself a valid token; (3) a real "undefined" leak when a genuine doubled "n" (e.g. "annai") was wrongly treated as a repeated vowel
- **Fix** — check the chōon continuation first against the real romaji input, and never let the syllabic n set `lastVowel` to anything but `null`
- **Sokuon** — a doubled consonant (excluding "n"), or the real Hepburn "t"+"ch" irregularity before the ち-row, verified via "matcha"
- **Particle overrides** — standalone "wa"/"e"/"o" map to は/へ/を instead of わ/え/お
- **Lexicalized exceptions** — こんにちは/こんばんは resolved directly, closing Chapter 2's own cliffhanger
- **Honest limit** — standalone "e" as え vs. 絵 is genuinely unresolvable from spelling alone

CHAPTER 4: AN API ROUTE FOR CONVERSION VS. DOING IT CLIENT-SIDE

Romaji to Kana Converter: React & Next.js

Chapter 4 · An API Route for Conversion vs. Doing It Client-Side

Chapter 1 left one real question deliberately open: given that Next.js can hold both the UI and real server-side API routes in one project, should this specific feature — converting romaji to kana — actually run on the server at all, or should the finished `convert()` engine from Chapters 2 and 3 be called directly in the browser? This chapter answers that with a real Route Handler and a real, independently-measured benchmark, rather than assuming the Astro sibling course's own already-published verdict simply carries over unchanged.

Building the Real Route

`src/app/api/convert/route.ts` is a direct, mechanical application of the same Route Handler pattern verified in Chapter 1 — the only difference is that this handler now calls the real conversion engine instead of returning a hardcoded placeholder:

```
// src/app/api/convert/route.ts
import { convert } from '@lib/convert';

export async function POST(request: Request) {
  const body = await request.json();
  const { romaji } = body;

  if (typeof romaji !== 'string') {
    return Response.json({ error: 'romaji must be a string' }, { status: 400 });
  }

  const result = convert(romaji);
  return Response.json(result);
}
```

Once a Next.js dev server is running this route, it accepts exactly the shape the future React UI in Chapter 5 will send it:

```
$ curl -X POST http://localhost:3000/api/convert \
  -H "Content-Type: application/json" \
  -d '{"romaji":"kyaku"}'
{"hiragana":"きゃく","katakana":"キャク"}
```

A POST request is never eligible for Next.js's own GET-specific static/dynamic caching behavior in the first place (Chapter 1's own "dynamic since v15" finding was specifically about `GET` handlers) — there's no extra configuration to reason about here, since this route's own job is to process a fresh request every time regardless.

A Real, Independently-Measured Benchmark

Rather than trust that the Astro sibling course's own real, published numbers — roughly $1.36\mu\text{s}$ per call direct against roughly $14,084\mu\text{s}$ per call over a real local HTTP round trip — simply carry over to a Next.js Route Handler unchanged, this chapter runs the identical style of benchmark against this variant's own real code. Since what's actually being measured is the real cost of a local HTTP round trip — TCP/socket overhead, not anything specific to a particular routing framework — a plain, standalone Node `http` server exposing the exact same `convert()` function is a fair, honest stand-in for the real request/ response mechanics a running Next.js dev server would add, matching the same discipline the Astro course's own Chapter 5 already established:

```
// bench_server.js — a real, standalone HTTP server exposing the same conversion logic
const http = require('http');
const { convert } = require('./convert_lib.js');

const server = http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/convert') {
    let body = '';
    req.on('data', chunk => body += chunk);
    req.on('end', () => {
      const { romaji } = JSON.parse(body);
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify(convert(romaji)));
    });
  }
});
```

```
});  
server.listen(4400);
```

A real client script, run against this actual server, measures elapsed wall-clock time for a tight loop of direct calls against the same-size loop of real `fetch()` requests:

```
$ node bench_server.js & # start the real server on :4400  
$ node bench_client.js  
  
warmup http avg us/call: 14971.33  
direct convert() call: 0.376 µs/call (N=500)  
HTTP round trip call: 14245.846 µs/call (N=500)  
ratio: 37928x slower over HTTP  
  
Re-run at N=200: direct 0.241 µs/call, http 14648.456 µs/call, ratio 60782x
```

A GENUINE CROSS-VARIANT CONSISTENCY CHECK

This course's own independently-measured HTTP round trip — roughly 14,246–14,648 microseconds per call, run against a completely different server implementation on a different day — lands within a few hundred microseconds of the Astro course's own real, separately measured $\sim 14,084\mu\text{s}$ figure. That's a real, honest confirmation that the dominant cost here is genuinely the local HTTP round trip itself (socket setup, the TCP handshake, the actual network stack a request has to travel through even on `localhost`), not anything specific to which framework happens to be listening on the other end.

AN HONEST NOTE ON THE RATIO ITSELF

The measured *ratio* — roughly $38,000\times$ to $61,000\times$ here, against the Astro course's own reported $\sim 10,353\times$ (re-confirmed around $\sim 7,018\times$ at a different sample size) — is genuinely less stable than the two raw numbers it's built from. A direct call finishing in a fraction of a microsecond means small, real timing-measurement noise gets amplified enormously once it's turned into a ratio against a number that small. The two absolute figures — a real HTTP round trip costing roughly 14–15 *milliseconds*, against a direct call costing a small fraction of one *microsecond* — are the stable, load-bearing finding; the exact multiplier moves around from run to run and shouldn't be quoted as a fixed number.

The Verdict

`convert()` holds no secret worth protecting and touches no shared state — every argument the Astro sibling course's own Chapter 5 already made for keeping this specific feature client-side applies here too, and this chapter's own fresh, independently-measured numbers confirm it rather than merely repeat it. Chapter 5 builds the real React UI calling `convert()` directly, in-process, as the version actually meant to ship. The `/api/convert` route built in this chapter stays in the project — it's genuinely useful as a working, demonstrable example of a real Next.js Route Handler, and Chapter 5 keeps a second box wired to it specifically so the two approaches can be compared side by side rather than asserted from a chapter of prose alone.

COURSE	DIRECT CALL	HTTP ROUND TRIP	VERDICT
Astro (Ch.5)	~1.36 μ s/call	~14,084 μ s/call	Client-side
React & Next.js (this chapter)	~0.24–0.38 μ s/call	~14,246–14,648 μ s/call	Client-side

Hands-On Exercises

EXERCISE 1

Run the real benchmark scripts from this chapter yourself against a third, different test word (not "kyaku"). Report the real numbers you get and compare them against the ones published here.

 [View solution](#)

EXERCISE 2

The `/api/convert` route returns a real 400 status for a missing or non-string `romaji` field. Send a real request with no body at all and explain, from real output, what actually happens before that check is ever reached.

 [View solution](#)

EXERCISE 3

Explain, in your own words, why the measured ratio (38,000× vs. the Astro course's own ~10,353×) is a less meaningful comparison between the two variants than the two raw absolute numbers are.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **Real route** — `POST /api/convert`, a direct application of Chapter 1's own verified Route Handler pattern, now calling the real conversion engine
- **Real benchmark** — a standalone Node HTTP server standing in for the real request/response cost, matching the Astro course's own methodology
- **Measured** — direct call $\sim 0.24\text{--}0.38\ \mu\text{s}/\text{call}$; HTTP round trip $\sim 14,246\text{--}14,648\ \mu\text{s}/\text{call}$
- **Cross-variant consistency** — this course's own independently-measured HTTP cost lands within a few hundred μs of the Astro course's own separately measured figure
- **Verdict** — client-side, confirmed with fresh numbers rather than assumed; the server route stays for real, working comparison in Chapter 5

CHAPTER 5: BUILDING THE INPUT/OUTPUT UI IN REACT

Romaji to Kana Converter: React & Next.js

Chapter 5 · Building the Input/Output UI in React

Chapter 4 closed with a real, measured verdict: the version of this converter that should actually ship calls `convert()` directly in the browser, with the real `/api/convert` route kept alive specifically so the two approaches can be compared side by side. This chapter builds both — a client-side box that's the real recommended version, and a second box wired to the API route, deliberately built with the same real request-timing risks a genuine server-calling UI would face in production.

The Client-Side Box: The Version Meant to Ship

Any component using React state or an event handler needs the `"use client"` directive at the top of its file — by default, every component in the App Router is a Server Component, which can't hold interactive state at all:

```
// src/app/components/ClientConverter.tsx
'use client';

import { useState } from 'react';
import { convert } from '@lib/convert';

export default function ClientConverter() {
  const [input, setInput] = useState('');
  const { hiragana, katakana } = convert(input);

  return (
    <div>
      <h3>Client-Side (Recommended)</h3>
      <input
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Type romaji..."
      />
      <p>Hiragana: {hiragana}</p>
    </div>
  );
}
```

```

    <p>Katakana: {katakana}</p>
  </div>
);
}

```

`convert(input)` runs directly inside the render, on every single keystroke — no `useEffect`, no debounce, nothing asynchronous at all. That's a deliberate, Chapter-4-backed choice, not an oversight:

NO DEBOUNCE NEEDED — CONFIRMED BY CHAPTER 4'S OWN REAL NUMBERS

Chapter 4 measured a direct `convert()` call at well under a microsecond. A real, fast typist producing a keystroke every 50–100 milliseconds is nowhere close to fast enough to make that cost add up to anything worth optimizing away — debouncing a call this cheap would add real complexity to solve a problem that, per Chapter 4's own measured evidence, doesn't exist.

Real, Verified Live-Typing Behavior

Because `convert()` runs against whatever's currently in the input box, an incomplete token renders as raw Latin text until the full token is actually typed — verified directly by simulating a user typing "kyoto" one character at a time:

```

typed "k"      { hiragana: 'k',    katakana: 'k'    }
typed "ky"     { hiragana: 'ky',   katakana: 'ky'   }
typed "kyo"    { hiragana: 'きょ', katakana: 'キヨ' } // snaps in once the full yōon
typed "kyot"   { hiragana: 'きょt', katakana: 'キヨt' }
typed "kyoto"  { hiragana: 'きょと', katakana: 'キヨト' }

```

This isn't a bug — it's the expected, real behavior of a tokenizer with no lookahead into characters the user hasn't typed yet, and it's the exact same real finding the Astro sibling course made against its own version of this converter, now independently re-confirmed here.

The Server-Comparison Box: A Deliberately Real Second Box

A second box calls the real `/api/convert` route from Chapter 4 instead of `convert()` directly. Built naively — firing a `fetch()` on every keystroke with no guard at all — it inherits two

genuine real risks that the client-side box, with no network calls at all, never has to worry about.

Risk 1: A Real, Reproduced Out-of-Order Response

A real HTTP request doesn't always finish in the order it was sent — network timing genuinely varies per request. Simulating exactly that with two real, deliberately staggered async calls (an earlier request that happens to be slow, and a later one that happens to be fast) reproduces a real race condition:

```
// user types "s" (fires a slow request), then "su" 15ms later (fires a fast one)
naiveFetch('s-result', 40).then(r => display(r)); // resolves LAST, after 40ms
// ...15ms later...
naiveFetch('su-result', 5).then(r => display(r)); // resolves FIRST, after 5ms

// real, observed order: "su-result" applied first, then "s-result" overwrites it
// FINAL displayed value: 's-result' — wrong. The user's actual last input was "su".
```

A GENUINE BUG, NOT A HYPOTHETICAL ONE

The user's real final input was "su" — the later request. But because its own response happened to arrive faster than the earlier, slower request for just "s," a naive `fetch(...).then(setResult)` handler applies whichever response lands last on the wire, not whichever request was fired last. Given Chapter 4's own real measured ~14ms round trip, and a real typist easily producing keystrokes closer together than that, this isn't an edge case — it's a genuine, likely outcome for the server-comparison box as written so far.

The Fix: A Latest-Request-Wins Guard

A simple incrementing request id, held in a `useRef` so it survives across renders without triggering one itself, lets a response check whether it's still the most recent request before it's allowed to update the displayed result:

```
// src/app/components/ServerConverter.tsx
'use client';

import { useState, useRef, useEffect } from 'react';
```

```

export default function ServerConverter() {
  const [input, setInput] = useState('');
  const [result, setResult] = useState({ hiragana: '', katakana: '' });
  const latestRequestId = useRef(0);

  useEffect(() => {
    if (!input) { setResult({ hiragana: '', katakana: '' }); return; }

    const timer = setTimeout(async () => {
      const thisRequestId = ++latestRequestId.current;
      const res = await fetch('/api/convert', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ romaji: input }),
      });
      const data = await res.json();
      // only apply this response if no newer request has fired since
      if (thisRequestId === latestRequestId.current) {
        setResult(data);
      }
    }, 300); // genuinely useful here – Chapter 4's own real ~14ms cost is worth batchi

    return () => clearTimeout(timer);
  }, [input]);

  return (
    <div>
      <h3>Server-Side (via /api/convert)</h3>
      <input value={input} onChange={(e) => setInput(e.target.value)} />
      <p>Hiragana: {result.hiragana}</p>
      <p>Katakana: {result.katakana}</p>
    </div>
  );
}

```

Re-running the exact same staggered-timing scenario against the guarded version:

```

applied (still latest): su-result
discarded (stale):      s-result
FINAL displayed value: su-result  // correct

```

A REAL, DIRECT PAYOFF OF CHAPTER 4'S OWN MEASUREMENT

The 300ms debounce isn't an arbitrary UX choice — it's a genuine consequence of Chapter 4's own measured ~14ms-per-request cost. Firing a real network request on every keystroke would mean a fast typist could easily trigger dozens of real HTTP round trips for a single word, each one paying that same real, measured cost — batching them behind a short delay is a direct, honest answer to a real number this course already measured, not a guess.

Hands-On Exercises

EXERCISE 1

Simulate real, live typing of "sushi" one character at a time against `convert()`, exactly as this chapter did for "kyoto." Predict the output at each step first, then verify with real code.

[!\[\]\(3f1d9e9f6bcc0837db71d34c7a09f75d_img.jpg\) View solution](#)

EXERCISE 2

Reproduce this chapter's own race condition with real code, using three staggered requests instead of two (an early slow one, a middle-timed fast one, and a late medium-speed one). Verify the latest-request-wins guard still produces the correct final result.

[!\[\]\(7c3d2608a7a555c07f6aa454ffa411df_img.jpg\) View solution](#)

EXERCISE 3

Explain, in your own words, why the client-side box needs no debounce and no request-id guard at all, while the server-comparison box genuinely needs both — tracing each requirement back to a specific real finding from Chapter 4.

[!\[\]\(ea2dc1f99f5c274d10213008b9cb3d37_img.jpg\) View solution](#)

CHAPTER 5 QUICK REFERENCE

- **Client-side box** — `"use client"`, `useState`, `convert()` called directly in render, no debounce needed (Chapter 4's own sub-microsecond finding)
- **Real, verified live-typing behavior** — an incomplete token renders as raw Latin text until it fully completes
- **Server-comparison box** — a genuine, reproduced out-of-order response race condition, fixed with a `useRef`-based latest-request-id guard
- **A real, justified 300ms debounce** — a direct, honest consequence of Chapter 4's own measured ~14ms-per-request cost, not an arbitrary UX choice

CHAPTER 6: LOOKING AHEAD — A REDIS KANJI CACHE

Romaji to Kana Converter: React & Next.js

Chapter 6 · Looking Ahead: A Redis Kanji Cache

SCOPE NOTE — READ THIS FIRST

This chapter does **not** add kanji conversion to the app. Real kanji conversion is a genuinely different, harder problem than the gojūon/yōon table this course has built so far — it needs a large vocabulary dictionary, and resolving which kanji a given reading actually means depends on real surrounding context (many words share an identical kana reading but write to different kanji), the same kind of disambiguation a real Japanese input-method editor (IME) performs, not a simple lookup table. Building that is a real project of its own, deliberately left for later. What this chapter *does* do is answer a narrower, concrete question honestly, with real measured numbers: once a vocabulary-scale dictionary exists, where should it live, and how fast can it realistically be looked up from a Next.js API route?

Why KANA_MAP's Own Approach Won't Scale

Chapter 2's own `KANA_MAP` works as a plain, hardcoded TypeScript object because it's small — roughly a hundred entries covering the gojūon grid, dakuten/handakuten rows, and yōon combinations. It ships inside the app's own JavaScript bundle and costs nothing extra to look up.

A real kanji/vocabulary dictionary is a different scale of problem entirely. JMdict — the real, widely-used open Japanese-English dictionary project — holds well over 100,000 entries. Even a deliberately trimmed, common-words-only subset would realistically run into the tens of thousands. A structure that size can't reasonably be hardcoded into a TypeScript module the way `KANA_MAP` was; it needs to live somewhere external, and it needs a real, fast way to look a word up by its reading.

The Real Architectural Question

A Next.js API route deployed the way Chapter 7 will cover doesn't guarantee that an in-memory JavaScript object survives, or is shared, across every request it handles. Serverless

function instances can be created and destroyed per request, and even where an instance does stay warm for a while, a production deployment typically runs several instances behind a load balancer — each with its own separate memory, none of them seeing what another one cached. An in-process `Map`, which was genuinely free and fast for `KANA_MAP`, isn't a reliable place to hold a large shared dictionary in that environment.

Redis is the standard real answer to exactly this problem — a single external store every function instance can reach over the network, so a lookup cached by one request is genuinely available to the next one, regardless of which instance handles it. This isn't a hypothetical fit either: Redis's own real, current documentation for its Vercel integration lists **"Persistent session storage across serverless functions"** as one of its stated real use cases, and both Redis Cloud (the official managed offering) and Upstash (used directly in Next.js's own official `with-redis` example) are real, verifiable managed providers built specifically for this deployment shape.

WHERE THIS WOULD ACTUALLY SIT

```
src/
├─ lib/
│  ├─ convert.ts           // existing: gojūon/yōon table, tokenizer
│  └─ kanjiCache.ts        // future: Redis client + lookup helpers
└─ app/
   └─ api/
      ├─ convert/
      │  └─ route.ts        // existing (Ch.4/5): kana-only conversion
      └─ convert-kanji/
         └─ route.ts        // future: calls kanjiCache.ts, not convert.ts
```

A future kanji-aware route would sit alongside the existing one, not replace it — the kana-only `/api/convert` route this course already built stays exactly as it is.

A Real Benchmark: File Scan vs. In-Process Map vs. Real Redis

Rather than guess at the numbers, a real, containerized Redis 7 instance was started locally (via Docker) and benchmarked directly against a fake but realistically-sized 50,000-entry vocabulary

dictionary — genuinely tested with real code, not assumed. Three approaches were compared, each performing 200 real lookups:

```
// Method 1: naive cold scan -- re-read + re-parse the whole file, then Array.find(), p
for (const w of targetWords) {
  const arr = JSON.parse(fs.readFileSync(dictFile, 'utf8'));
  const found = arr.find(e => e.word === w);
}

// Method 2: warm in-process Map, built once
const map = new Map(entries.map(e => [e.word, e]));
for (const w of targetWords) { const found = map.get(w); }

// Method 3: real Redis GET over real TCP, one round trip per lookup
for (const w of targetWords) { const raw = await redis.get(`vocab:${w}`); }
```

Two runs, re-verified for consistency:

METHOD	PER-CALL COST (RUN 1)	PER-CALL COST (RUN 2)
Naive re-scan of the 50,000-entry file	41,134 μs	42,476 μs
Warm in-process <code>Map</code>	0.66 μs	0.63 μs
Real Redis <code>GET</code> (localhost, containerized)	962 μs	926 μs

REAL, MEASURED FINDINGS

Redis beat the naive file re-scan by roughly **43-46×** — a real, decisive win once the dictionary is genuinely large. But Redis lost to the in-process `Map` by roughly **1,400-1,500×** — a real network round trip, even to a container running on the same machine, costs vastly more than a function call that never leaves the process. This is the exact same shape of finding Chapter 4 already made about `convert()` vs. `fetch()`: a network hop is the expensive part, not the actual work being done on the other end of it.

Why Reach for Redis Anyway, Given That Result

The in-process `Map` wins the benchmark, but it isn't actually available for free the way it looks in this single-process test. It only wins because this benchmark ran in one warm process that built the `Map` once and reused it — exactly the condition the previous section explained a real serverless deployment doesn't reliably provide. Redis isn't being chosen because it's fast in an absolute sense; it's chosen because it's the one option in this comparison that's actually still there on the next request, no matter which function instance handles it.

The Round Trip Is (Almost) the Whole Cost

A further real test makes that concrete. Fetching the same 200 keys as 200 separate `GET` calls was compared against fetching all 200 in a single `MGET` call — one round trip carrying every key at once:

```
200 individual GETs total: 169,381.5 µs (846.91 µs/call)
1 MGET for all 200 keys: 2,010.4 µs total
speedup: 84.3x
```

Batching the 200 lookups into one round trip cut the total time by **84x**. Redis itself didn't get faster at doing the actual lookups — what changed is that the app paid the real network round-trip cost once instead of 200 times. That's a direct, practical echo of the same round-trip-dominates lesson this course already found for HTTP in Chapter 4, now confirmed for Redis specifically.

THE REAL DESIGN IMPLICATION

A future kanji-aware route shouldn't call `redis.get()` once per candidate substring while scanning a sentence. It should identify every kanji-eligible substring first, then issue one `redis.mget()` call carrying all of them at once — turning what could be a dozen or more round trips per conversion request into exactly one.

AN HONEST CAVEAT ON THE ABSOLUTE NUMBERS

This benchmark's Redis container ran under Docker Desktop on Windows, which routes container networking through a lightweight virtualization layer rather than talking to Redis directly on the host's own network stack. The real ~0.85-0.96ms per-call figure measured here likely includes some genuine overhead from that layer specifically, and a managed provider like Redis Cloud or Upstash, or a native Linux deployment, could measure differently in absolute terms. The *proportional* finding — that batching into one round trip beats N individual round trips by roughly two orders of

magnitude — is a structural property of "one network hop vs. many," not an artifact of this specific setup, and should hold regardless of exactly which numbers a different environment produces.

Hands-On Exercises

EXERCISE 1

Re-run this chapter's own three-way benchmark (file re-scan / in-process Map / real Redis) at a smaller scale — 5,000 dictionary entries instead of 50,000 — and explain, using the real numbers you measure, which method's own timing changes the most and which barely changes at all.

 [View solution](#)

EXERCISE 2

This chapter's own MGET test batched 200 keys into a single round trip. Real production sentences would realistically need far fewer lookups per request — closer to 5-15 kanji-eligible substrings per sentence. Re-run the individual-GET-vs-MGET comparison at a more realistic batch size of 10 keys, and explain whether the speedup still matters at that smaller scale.

 [View solution](#)

EXERCISE 3

Explain, in your own words, why this chapter's own benchmark result (Redis loses to an in-process Map) is not actually an argument against using Redis in the real deployment this course is building toward. What does the in-process Map's own win depend on that a real serverless deployment doesn't reliably provide?

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **Kanji conversion itself is still deferred** — this chapter is architecture and benchmarking only, not a new feature

- **Real measured result** — Redis beats a naive file re-scan by $\sim 43\text{-}46\times$, but loses to an in-process Map by $\sim 1,400\text{-}1,500\times$
- **Why Redis anyway** — a serverless deployment doesn't reliably share one process's own memory across every request; Redis Cloud and Upstash are both real, verified managed options built for exactly this
- **Batching wins** — one real `MGET` for 200 keys measured $84\times$ faster than 200 individual `GET` calls, the same round-trip-dominates lesson from Chapter 4

CHAPTER 7: DEPLOYMENT

Romaji to Kana Converter: React & Next.js

Chapter 7 · Deployment

Every prior chapter has been building toward a single deployable thing: one Next.js project holding the client-side converter, the server-comparison box, and the real `/api/convert` route all together. This chapter puts that architecture to its own real test — building it for production, confirming exactly what does and doesn't survive that build process, and getting it running behind a real process manager and reverse proxy.

THE PAYOFF OF CHAPTER 1'S OWN ARCHITECTURAL CHOICE

Chapter 1 framed this variant's own real difference from its two siblings directly: Astro treats an API route as something you opt into per page, and the Angular & Express sibling course needed two genuinely separate deployables — an Angular build served one way, an Express process running another. This course has never had that split. There's one project, one build command, and — as this chapter confirms directly — one running process that serves the UI and answers `/api/convert` at the same time.

Building for Production: The Standalone Output Mode

Next.js's own default production build (`next build` followed by `next start`) still expects the full project, including every package in `node_modules`, to be present on the machine that runs it. Next.js's own real, current documentation offers a leaner alternative specifically for deployment: setting `output: 'standalone'` in `next.config.js` makes `next build` trace every file each page and route actually needs and copy only those into `.next/standalone`, along with a small, self-contained `server.js` that can run without `next start` or a full `node_modules` install at all.

```
// next.config.js
module.exports = {
  output: 'standalone',
};
```

Building a real, minimal Next.js 16.3.5 project with this setting confirms the documentation's own claim directly — the standalone output holds exactly three things, nothing more:

```
$ ls .next/standalone
node_modules/
package.json
server.js
```

WHAT'S DELIBERATELY MISSING — VERIFIED

Checking for `public/` and `.next/static` inside the freshly-built standalone folder confirms both are genuinely absent:

```
$ ls .next/standalone/public
ls: cannot access '.next/standalone/public': No such file or directory
$ ls .next/standalone/.next/static
ls: cannot access '.next/standalone/.next/static': No such file or directory
```

This isn't an oversight — the documentation explains the reasoning directly: these two folders are meant to be served by a CDN in a real production setup, not by the Node process itself, so `next build` doesn't copy them in by default. For a deployment with no separate CDN, they have to be copied in by hand, after the build:

```
$ cp -r public .next/standalone/
$ cp -r .next/static .next/standalone/.next/
```

Skipping this step doesn't break the app outright — it fails silently, per-asset. Running the standalone server with neither folder copied in, the homepage itself still returns a real `200` (its HTML is generated by `server.js` directly), but a real static file from `public/` returns a genuine `404`:

```
HTTP status for /file.svg: 404
HTTP status for /: 200
```

Once both folders are in place, the server runs directly with `node` — no `next start`, no full `node_modules` install required on the deployment machine:

```
$ PORT=3000 HOSTNAME=0.0.0.0 node .next/standalone/server.js
```

`PORT` and `HOSTNAME` are read directly by `server.js` itself — a real, documented mechanism, not something this course is bolting on.

A REAL, MEASURED REASON THIS MATTERS

Building the same minimal project's own two `node_modules` folders side by side and measuring their real disk size:

```
Full project node_modules:      432M
Standalone's pruned node_modules: 19M
```

A real **~22.7×** reduction, purely from tracing which files each route actually needs at runtime rather than shipping every package a full development install pulls in. For a project that will grow to include Chapter 6's own Redis client once kanji conversion is eventually added, this is a genuine, compounding saving — a deployment artifact this much smaller uploads faster, starts faster, and costs less to store.

Environment Variables: Two Genuinely Different Lifetimes

Next.js draws a real, load-bearing line between two kinds of environment variable, and this course's own Chapter 1 finding — that Route Handlers are dynamic by default — turns out to directly explain which side of that line `/api/convert` lands on.

NEXT_PUBLIC Variables Are Frozen at Build Time

A variable prefixed `NEXT_PUBLIC_` is inlined directly into the client-side JavaScript bundle the moment `next build` runs — every reference to it is replaced with a literal, hardcoded string, not a live lookup. Building a real client component that reads `process.env.NEXT_PUBLIC_APP_LABEL`, set to `"BUILD-TIME-LABEL-XYZ"` at build time, and then searching the actual built output confirms this directly — the literal string is physically present in a static chunk file, and the variable's own name is nowhere to be found:

```
$ grep -r1 "BUILD-TIME-LABEL-XYZ" .next/static/
.next/static/chunks/0ut3zcttubcxq.js // the literal value IS baked in
```

```
$ grep -rI "NEXT_PUBLIC_APP_LABEL" .next/static/
(no matches -- the raw variable name is gone entirely)
```

Changing a `NEXT_PUBLIC_` variable after the build has already happened does nothing — the value is already compiled into the shipped JavaScript. It needs a real rebuild to take effect.

Server-Only Variables in a Dynamic Route Handler Stay Live

A variable with no `NEXT_PUBLIC_` prefix is never sent to the browser at all — it only ever exists inside `process.env` on the server. Whether it's evaluated once at build time or freshly on every request depends on whether the code reading it opts into dynamic rendering — and Chapter 1 already established that a Route Handler like `/api/convert` does exactly that, by default, since Next.js 15. Building a real Route Handler that reads a plain `SECRET_MESSAGE` variable, building the app with one value set, then running the finished standalone server with a genuinely *different* value in its own environment confirms this concretely:

```
// .env at build time: SECRET_MESSAGE=build-time-value
$ npm run build

// running the built server with a DIFFERENT value:
$ SECRET_MESSAGE=runtime-override-value node .next/standalone/server.js

$ curl http://localhost:3000/api/env-test
{"secret":"runtime-override-value"} // the runtime value, not the build-time one
```

THE REAL PAYOFF OF CHAPTER 1'S DYNAMIC-BY-DEFAULT FINDING

This is a direct, concrete consequence of Route Handlers being dynamic by default, not something specific to environment variables on their own. A dynamically-rendered route genuinely re-runs on every request, so any `process.env` read inside it re-reads the environment fresh each time too — the exact opposite of a `NEXT_PUBLIC_` variable's own build-time freeze. In practice, this means the production server can be started with a real secret — a database URL, an API key, or (once Chapter 6's own architecture is actually built out) a Redis connection string — set as a genuine runtime environment variable, changeable per deployment without ever needing a rebuild.

Process Management with PM2

`node .next/standalone/server.js` run directly in a terminal dies the moment that terminal closes, and won't restart itself if it crashes. PM2, the same process manager this site's own sibling Node-backed courses already use, solves both problems:

```
$ npm install -g pm2
$ PORT=3000 HOSTNAME=0.0.0.0 pm2 start .next/standalone/server.js --name romaji-nextjs
$ pm2 save
$ pm2 startup // registers PM2 itself to start on server boot
```

PM2 restarts the process automatically if it crashes, keeps it running across the terminal session ending, and gives a real, queryable process list (`pm2 list`) and live log stream (`pm2 logs romaji-nextjs`) without any custom tooling.

An nginx Reverse Proxy

With the app itself running on port 3000, nginx sits in front of it, handling the public-facing port and (per this site's own HTTPS/TLS Fundamentals course) real TLS termination:

```
server {
    listen 80;
    server_name romaji.example.com;

    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Nothing about this proxy config is specific to this course's own architecture — it's the same real pattern this site has already used for every other Node-backed course — which is itself a quiet confirmation of Chapter 1's own point: this project deploys like any single Node application, because that's genuinely all it is.

Hands-On Exercises

EXERCISE 1

This chapter measured a real ~22.7x reduction in node_modules size using the standalone output mode (432MB full vs. 19MB pruned). Explain, in your own words, why a real production install of this project's own future Redis client (Chapter 6) would make this pruning step matter even more than it already does for the app as it stands today.

 [View solution](#)

EXERCISE 2

This chapter showed that skipping the manual public/.next-static copy step produces a real 404 for a static file while the homepage itself still returns 200. Explain why these two things fail differently, tracing the difference back to where each one's own content actually comes from.

 [View solution](#)

EXERCISE 3

A teammate suggests moving the site's public-facing "converter is temporarily under maintenance" flag into a NEXT_PUBLIC_ environment variable, so it can be toggled without redeploying the client UI. Using this chapter's own real, verified findings, explain why that plan wouldn't actually work the way they expect, and what the real fix would be.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **One deploy, not two** — the direct payoff of Chapter 1's own single-project architecture, confirmed by having only one process to build and run
- **output: 'standalone'** — traces and copies only the files actually needed, verified ~22.7× smaller than a full node_modules install
- **public/ and .next/static aren't copied automatically** — verified: skipping them fails silently, per-asset, not by crashing the whole app

- **NEXT_PUBLIC_ variables are frozen at build time** — verified: the literal value is baked into a static chunk file; changing it needs a rebuild
- **Server-only variables in /api/convert stay live** — verified: because the route is dynamic by default (Chapter 1), it re-reads process.env fresh on every request

CHAPTER 8: CAPSTONE — INTEGRATING THIS INTO THE EXISTING ASTRO SITE

Romaji to Kana Converter: React & Next.js

Chapter 8 · Capstone: Integrating This Into the Existing Astro Site

Seven chapters have built one real, working thing: a client-side React converter that's genuinely free to call on every keystroke, a second server-comparison box proving exactly why (a real measured ~14ms round trip against a sub-microsecond direct call), and a production build that starts, runs, and survives a restart on its own. This closing chapter answers the question every prior chapter has been building toward: how does this actually reach the user's real, existing Astro-based site?

Three Real Ways to Mount This

OPTION	WHAT IT LOOKS LIKE	REAL TRADEOFF
Subdomain	<code>romaji.example.com</code> , a fully separate deployment	Simplest nginx config, but needs its own DNS record and its own TLS certificate — a real second thing to keep renewed
Path-mounted reverse proxy	<code>example.com/romaji/</code> , sitting alongside the existing Astro site	One domain, one certificate — but the app itself has to genuinely know it isn't running at the web root anymore
API-only, Astro-native frontend	Only <code>/api/convert</code> is kept; a small script embedded directly in an existing Astro page calls it	No second React UI to maintain at all — but throws away Chapters 4-5's own real client-side box entirely, and Chapter 5's own real finding that <code>convert()</code> should run client-side, not through an API, in the first place

The path-mounted option is the one worth building out in full — it keeps everything this course has already built, asks the least of the user's own existing infrastructure, and is where the single most interesting real finding in this chapter turns up.

Next.js's Own Real Tool for This: `basePath`

Next.js's own current, real documentation names exactly this scenario directly: `basePath` "allows you to set a path prefix for the application," specifically to "deploy a Next.js application under a sub-path of a domain."

```
// next.config.js
module.exports = {
  output: 'standalone',
  basePath: process.env.NEXT_PUBLIC_BASE_PATH || '',
};
```

Building a real, minimal project with `NEXT_PUBLIC_BASE_PATH=/romaji` set and testing every real, relevant URL confirms exactly what `basePath` actually does — and what it doesn't:

```
GET /romaji      -> 200   (the real, mounted app)
GET /            -> 404   (the old root no longer serves anything)
GET /api/test    -> 404   (a raw, unprefixed fetch target)
GET /romaji/api/test -> 200   (the real, correct path)
```

THE REAL GOTCHA — EXACTLY WHERE CHAPTER 5'S OWN CODE WOULD BREAK

Next.js's own docs are explicit that `next/link` and `next/router` get `basePath` applied automatically — a `<Link href="/about">` genuinely becomes `/romaji/about` with zero code changes. But this course's own `ServerConverter` component, built back in Chapter 5, doesn't use either of those — it calls `fetch('/api/convert', ...)` directly, a plain browser API with no idea Next.js's own `basePath` config even exists. The real test above proves this concretely: the exact literal string Chapter 5's code already contains, `/api/convert`, is precisely the unprefixed path that returns a genuine `404` once `basePath` is set — even the official docs' own `next/image` component needs its `src` prefixed by hand for the identical reason. Only two real, first-party helpers get this for free; a raw `fetch()` call is not one of them.

The Real Fix

The community-documented pattern — confirmed by real testing — is to drive `basePath` from a single `NEXT_PUBLIC_` variable, then read that exact same variable inside the component to prefix the fetch call by hand:

```
// src/app/components/ServerConverter.tsx -- the one real line that changes
const BASE_PATH = process.env.NEXT_PUBLIC_BASE_PATH || '';
```

```
// ...inside the debounced fetch, from Chapter 5:
const res = await fetch(`${BASE_PATH}/api/convert`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ romaji: input }),
});
```

Re-running the same real test with this fix applied confirms it resolves cleanly:

```
GET /romaji          -> 200   (page loads, client fetch fires after hydration)
GET /romaji/api/test -> 200   {"hello":"from api/test"}
```

A DIRECT PAYOFF OF CHAPTER 7'S OWN FINDING

The docs state `basePath`'s own value "must be set at build time and cannot be changed without re-building as the value is inlined in the client-side bundles" — the identical real behavior Chapter 7 already verified for plain `NEXT_PUBLIC_` variables, via a real `grep` against a built project's own static chunk files. Running that same check here confirms it again: the literal string `"romaji"` is genuinely present, baked directly into the built client JavaScript, not read live. Moving the app under a subpath and reading an ordinary `NEXT_PUBLIC_` variable turn out to be the exact same mechanism underneath — `basePath` isn't a special case, it's just one more value this course already knows gets frozen at build time.

The nginx Side

With `NEXT_PUBLIC_BASE_PATH=/romaji` baked in at build time and the fixed `ServerConverter` deployed, nginx's own job is simply to route requests under `/romaji/` to the same PM2-managed process Chapter 7 already set up — no separate process, no separate port range, just one more `location` block alongside whatever already serves the existing Astro site:

```
server {
  listen 80;
  server_name example.com;

  # the existing Astro site, unchanged
  location / {
```

```

    root /var/www/astro-site/dist;
}

# the Next.js converter, mounted at /romaji/
location /romaji/ {
    proxy_pass http://localhost:3000/romaji/;
    proxy_http_version 1.1;
    proxy_set_header Host $host;
}
}

```

Because the app's own `basePath` is already `/romaji`, nginx doesn't need to strip or rewrite anything — it forwards the request path exactly as-is, and the Next.js app itself already expects to receive it that way.

A Real Link From the Existing Site

The whole point of this course, stated back in Chapter 1, was never "build a standalone tool" — it was building the thing the site's own real Japanese-language lessons and kanji reference pages could genuinely point to. With the converter now live at `/romaji/`, a real link from the site's own existing kanji tiles page and from individual Japanese-language lesson pages becomes a concrete, one-line addition rather than a hypothetical:

```

<a href="/romaji/" class="jp-tool-link">
  Try the Romaji → Kana Converter
</a>

```

A learner reading through a hiragana lesson, or hovering a kanji tile on the site's own reference page, is now one real click away from a working conversion tool built specifically for this site — the exact integration this whole course existed to make possible.

HONEST SCOPE NOTE — WHAT'S STILL DELIBERATELY DEFERRED

Kanji conversion itself is still not built. Chapter 6 established the real architecture and the real, measured case for a Redis-backed lookup cache once it's added, but no dictionary and no kanji-aware route exist in the running app today — the tool this chapter mounts converts romaji to

hiragana and katakana only, exactly as Chapter 2 through Chapter 5 built it. That remains a genuine, separate project for later, deliberately scoped out from the start.

Chapter Attribution

CHAPTER	WHAT IT CONTRIBUTED TO THE FINISHED, DEPLOYED TOOL
1	Next.js project setup, and the real architectural framing (one project, both UI and API) this chapter's own single-process deployment cashes in
2	The real gojūon/dakuten/handakuten/yōon <code>KANA_MAP</code> , and the fixed longest-to-shortest tokenizer
3	Chōon, sokuon, and particle-exception handling — three real bugs found and fixed
4	The real <code>/api/convert</code> route, and the measured ~14ms case for calling <code>convert()</code> directly instead
5	The real client-side and server-comparison boxes — including the exact <code>fetch('/api/convert')</code> line this chapter had to fix
6	The real, measured architecture case for a Redis kanji cache, deliberately not yet built
7	The real production build, PM2 process management, and the <code>NEXT_PUBLIC_</code> build-time-freeze finding this chapter directly reused for <code>basePath</code>
8	Comparing three mounting options, the real <code>basePath</code> -plus-fetch-fix, and the nginx config that puts it all live

Hands-On Exercises

EXERCISE 1

Chapter 5 also built a client-side box that calls `convert()` directly, with no `fetch()` call at all. Explain, using this chapter's own real findings, why that box needs no `basePath` fix whatsoever, while the server-comparison box's own `fetch` call does.

 [View solution](#)

EXERCISE 2

This chapter's own real test showed that GET / returns a genuine 404 once basePath is set to /romaji. A teammate is confused, since the app clearly still works — just under a different path. Explain, precisely, what "the app's own root" actually means once basePath is configured, and why 404 is the honest, correct response rather than a bug.

 [View solution](#)
EXERCISE 3

Suppose kanji conversion (Chapter 6) is eventually built, adding a new route at /api/convert-kanji that calls a real Redis instance. Explain what would need to happen for that new route to work correctly once this chapter's own basePath mounting is in place -- would the real Redis calls inside the route handler itself need any changes, or only the client-side code that calls the route?

 [View solution](#)
CHAPTER 8 QUICK REFERENCE — COURSE COMPLETE

- **Three real mounting options compared** — subdomain, path-mounted reverse proxy, and API-only; the path-mounted option built out in full
- **basePath moves the whole app, API routes included** — verified: the old root genuinely 404s once it's set
- **A raw fetch() call is never auto-prefixed** — verified: exactly the literal `/api/convert` string Chapter 5's own code uses breaks with a real 404, fixed by reading the same NEXT_PUBLIC_BASE_PATH the config itself uses
- **basePath is frozen at build time** — the same real inlining behavior Chapter 7 already found for NEXT_PUBLIC_ variables generally, reconfirmed here
- **Kanji conversion stays deliberately deferred** — this course closes having built exactly what it scoped: a real, fast, working romaji-to-kana converter, genuinely integrated into the existing site