

PERSONAL MULTI-TECHNOLOGY PROJECTS

Romaji to Kana Converter: Astro

A real, working romaji-to-kana converter built in the site's own real framework — a verified gojuon/dakuten/yoan mapping table, real long-vowel/sokuon/particle edge-case handling, a live client-side UI, a real measured client-vs-server latency benchmark, styling reused from the site's own established tool palette, a full standalone deployment, and a capstone comparing two real integration paths onto the live site itself — the most direct "could genuinely ship as a new page tomorrow" variant of the three-course Romaji to Kana Converter set.

8 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & WHY THIS FITS DIRECTLY ONTO THE EXISTING SITE

Romaji to Kana Converter: Astro

Chapter 1 · Project Overview & Why This Fits Directly Onto the Existing Site

This is one of three courses building the exact same tool in three genuinely different architectures — a web app that takes romaji (Latin-alphabet-transliterated Japanese) text and converts it into hiragana and katakana. **Romaji to Kana Converter: Angular & Express** is already complete, and committed to a real, dedicated Express API from its own Chapter 5 onward; **Romaji to Kana Converter: React & Next.js** remains outlined. This course — the Astro variant — takes the most direct real angle of the three: the site this converter is actually meant to join is itself already built in Astro, so this isn't a hypothetical "how would you build this in Astro" exercise so much as "here's what adding this directly to the real, live site would actually look like."

What the App Actually Does

- **Convert romaji text** — typed Latin-alphabet Japanese, like `konnichiwa` — into both hiragana (こんにちは) and katakana (コンニチワ).
- **Handle real Japanese-specific edge cases** correctly — long vowels, the small つ (sokuon, doubled consonants), and particle exceptions (は pronounced "wa," へ pronounced "e") — covered in real depth in Chapter 3.
- **Ties directly into content already on this site** — the existing Japanese-language lessons and the kanji tiles/reference-materials pages, both real, immediate practical uses for a working romaji-to-kana tool once it's live, and the real focus of Chapter 8's own capstone.

Full kanji conversion is explicitly out of scope for this course — a genuinely bigger, real dictionary-and-context-dependent problem (the same word in kana can map to several different kanji depending on meaning), deliberately deferred as a real future project of its own rather than something this course pretends to solve as a side effect of romaji conversion.

Why Astro for This One

Of the three variants in this trio, this is the one built specifically to answer "could this genuinely just be a page on my real site" — and the answer here is a real, literal yes, not a metaphor:

- **Astro is already the real, live framework this site runs on.** Adding this converter later isn't a port from one framework to another — it's adding one more page to a project that already exists, using exactly the same tools every other page on the site was already built with.
- **No database needed at all, for the core MVP.** Converting romaji to kana is a pure, stateless function — type something in, get an answer back, nothing to remember between visits and nothing to persist. That's a genuine, deliberate contrast with this site's own Premier League Predictor: Astro course, which needs a real SQLite database from its own Chapter 1 precisely because a season of fixtures, predictions, and league standings *does* need to persist — see the finding below.
- **Astro's own component model gives this just enough interactivity without deciding anything upfront.** A form and a script can run entirely client-side inside a normal static Astro page, with no special configuration at all — which is exactly what keeps Chapter 5's own client-vs-server comparison a real, open question rather than one this chapter quietly settles by how the project gets set up.

COURSE	HOW THE CLIENT-VS-SERVER QUESTION IS TREATED
Romaji to Kana Converter: Astro (this course)	Genuinely open — Chapter 5 is a real, dedicated comparison: "Client-Side vs. Server-Side Conversion: Where Should the Logic Actually Run?"
Romaji to Kana Converter: React & Next.js	Also open — that course's own Chapter 4 compares "An API Route for Conversion vs. Doing It Client-Side"
Romaji to Kana Converter: Angular & Express	Decided from the outset — a real, dedicated Express API from Chapter 5 onward, since giving Angular a paired backend it doesn't otherwise have anywhere on this site is that course's own reason for existing

Unlike the Angular course, this one has no structural reason to decide the question early — Astro can genuinely do either approach well, so there's nothing forcing a choice before the real evidence is in. Chapter 5 builds both versions for real and compares them directly, rather than asserting one is better.

THIS COURSE ASSUMES REAL ASTRO GROUNDWORK ALREADY COVERED ELSEWHERE

This site's own [astro1](#) (Astro) course already covers Astro's own component model, file-based routing, and static-first rendering philosophy in real depth, and [website-rebuild-with-astro1](#)

covers a full real Astro site rebuild. This course doesn't re-teach any of that — it assumes it, and moves straight into this converter's own real mapping table, edge cases, and UI.

Setting Up: A Plain Astro Project, No Database, No Adapter Yet

A local Node.js install is assumed from here on. Scaffold a fresh Astro project first:

```
# Scaffold a new Astro project (choose the empty template, TypeScript "Strict")
npm create astro@latest -- romaji-converter-astro
cd romaji-converter-astro
npm run dev
```

Visiting `http://localhost:4321` should show Astro's own default starter page — confirmation the project itself is set up correctly, before a single line of conversion logic exists.

Astro renders every page statically by default, and this project is deliberately left in that default state for now. Verified directly against Astro's own current documentation: a freshly scaffolded project always starts in `output: 'static'` mode, with the entire site pre-rendered to plain HTML at build time — and that default is genuinely correct here, not just left unexamined, since nothing about romaji-to-kana conversion actually requires a server at all yet:

```
// astro.config.mjs
import { defineConfig } from 'astro/config';

export default defineConfig({});

// No output mode set, no adapter installed – 'static' is the real default,
// and there's nothing yet in this project that needs anything else.
```

A minimal project structure, following this site's own established Astro project convention rather than inventing a new one — deliberately without a `lib/db.ts` or a `data/` folder, since this app has nothing to persist:

```
romaji-converter-astro/
├─ astro.config.mjs
├─ src/
│   └─ lib/
```

```
| | └─ convert.ts      # the real romaji-to-kana mapping table (Chapter 2-3)
| └─ pages/
| | └─ index.astro    # the input/output UI itself (Chapter 4)
| └─ components/
└─ public/
```

A minimal placeholder page confirms the setup works before any real conversion logic exists:

```
---
// src/pages/index.astro
---
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Romaji to Kana Converter</title>
  </head>
  <body>
    <h1>Romaji to Kana Converter</h1>
    <p>Setup confirmed – the real conversion engine starts in Chapter 2.</p>
  </body>
</html>
```

Reloading `http://localhost:4321` should now show this exact page — a real, working Astro project, statically rendered, with nothing installed or configured beyond what a fresh scaffold already provides.

A REAL, DELIBERATE CONTRAST WITH THIS COURSE'S OWN ASTRO SIBLING

Premier League Predictor: Astro sets `output: 'server'` and installs the Node adapter in its own very first chapter, because that app genuinely needs one — a season's worth of fixtures, predictions, and league standings has to persist between visits, and persisting anything at all means some part of the app has to run on a server. This converter has no equivalent need: given the same romaji input, it produces the same kana output every single time, with nothing to remember in between. Leaving the project in Astro's own default static mode isn't a shortcut taken here — it's the genuinely correct choice for what this app actually does, and Chapter 5 is where that choice gets tested directly against a real server-rendered alternative rather than just assumed to be right.

Where This Course Is Headed

A real romaji-to-kana mapping table (Chapter 2); the genuinely tricky edge cases — long vowels, sokuon, and particle exceptions (Chapter 3); building the actual input/output UI in Astro (Chapter 4); the real client-vs-server comparison this chapter deliberately left open (Chapter 5); styling this to actually match the existing site's own look (Chapter 6); deployment (Chapter 7); and a capstone shipping this as a real new page on the live site, closing with an honest look at kanji conversion as a genuine future sub-project of its own (Chapter 8).

Hands-On Exercises

EXERCISE 1

Scaffold the project exactly as this chapter describes, confirm `npm run dev` serves the placeholder page at Astro's own default port, and inspect the real generated `astro.config.mjs` — confirm the default output mode and that no adapter appears anywhere in it yet.

 [View solution](#)

EXERCISE 2

In your own words, explain why this course treats the client-vs-server conversion question as something to explore across Chapter 5 — the same way its still-outlined Next.js sibling does — while its completed Angular & Express sibling decided that question up front, and explain why this course's own MVP needs no database at all where its Premier League Predictor: Astro sibling does.

 [View solution](#)

EXERCISE 3

Without installing any adapter, add `export const prerender = false;` to the top of a scratch page in your own project and run a real production build with `npm run build`. Observe what actually happens, and explain — tying it back to this chapter's own finding — exactly why on-demand rendering can't work without one.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **The shared app** — romaji-to-hiragana/katakana conversion, kanji conversion deliberately deferred
- **Why this variant** — the real site this converter is meant to join is already built in Astro, making this the most direct "just add a page" story of the trio
- **Open, not decided** — unlike its Angular & Express sibling, this course treats client-vs-server conversion as a real, dedicated Chapter 5 comparison rather than committing upfront
- **No database for the MVP** — conversion is a pure, stateless function; a genuine, deliberate contrast with this site's own Premier League Predictor: Astro sibling
- **Setup** — a fresh Astro project left in its own real default `output: 'static'` mode, no adapter installed
- **Assumed groundwork** — Astro fundamentals, covered in this site's own dedicated `astro1` and `website-rebuild-with-astro1` courses
- **Next chapter:** The Conversion Engine — a real romaji-to-kana mapping table

CHAPTER 2: THE CONVERSION ENGINE: A REAL ROMAJI-TO-KANA MAPPING TABLE

Romaji to Kana Converter: Astro

Chapter 2 · The Conversion Engine

Chapter 1 deliberately left the client-vs-server question open for Chapter 5 to answer with real evidence. Until then, nothing in this project needs to know where the conversion logic will eventually run — it just needs to be correct. This chapter builds `src/lib/convert.ts` as a plain, framework-agnostic module, tested directly with `node`, with no Astro page, component, or endpoint touching it yet.

The Gojūon Table: Japan's Real 5×10 Sound Grid

Japanese kana are built around a real, systematic structure — five vowels (a, i, u, e, o) crossed with nine consonant rows, plus one standalone sound (ん/ン, "n"). A handful of cells are irregular under Hepburn romanization — し is "shi" not "si," ち is "chi" not "ti," つ is "tsu" not "tu," ふ is "fu" not "hu" — genuine historical exceptions baked into the standard system itself, not something this course invented:

```
// src/lib/convert.ts – the base gojūon grid
export const KANA_MAP: Record<string, { hiragana: string; katakana: string }> = {
  // Vowels
  a: { hiragana: 'あ', katakana: 'ア' },
  i: { hiragana: 'い', katakana: 'イ' },
  u: { hiragana: 'う', katakana: 'ウ' },
  e: { hiragana: 'え', katakana: 'エ' },
  o: { hiragana: 'お', katakana: 'オ' },

  // K row
  ka: { hiragana: 'か', katakana: 'カ' }, ki: { hiragana: 'き', katakana: 'キ' },
  ku: { hiragana: 'く', katakana: 'ク' }, ke: { hiragana: 'け', katakana: 'ケ' },
  ko: { hiragana: 'こ', katakana: 'コ' },

  // S row – shi is the real Hepburn exception, not "si"
  sa: { hiragana: 'さ', katakana: 'サ' }, shi: { hiragana: 'し', katakana: 'シ' },
  su: { hiragana: 'す', katakana: 'ス' }, se: { hiragana: 'せ', katakana: 'セ' },
  so: { hiragana: 'そ', katakana: 'ソ' },
```

```

// T row – chi and tsu are the real exceptions
ta: { hiragana: 'た', katakana: 'タ' }, chi: { hiragana: 'ち', katakana: 'チ' },
tsu: { hiragana: 'つ', katakana: 'ツ' }, te: { hiragana: 'て', katakana: 'テ' },
to: { hiragana: 'と', katakana: 'ト' },

// N row
na: { hiragana: 'な', katakana: 'ナ' }, ni: { hiragana: 'に', katakana: 'ニ' },
nu: { hiragana: 'ぬ', katakana: 'ヌ' }, ne: { hiragana: 'ね', katakana: 'ネ' },
no: { hiragana: 'の', katakana: 'ノ' },

// H row – fu is the real exception, not "hu"
ha: { hiragana: 'は', katakana: 'ハ' }, hi: { hiragana: 'ひ', katakana: 'ヒ' },
fu: { hiragana: 'ふ', katakana: 'フ' }, he: { hiragana: 'へ', katakana: 'ヘ' },
ho: { hiragana: 'ほ', katakana: 'ホ' },

// M row
ma: { hiragana: 'ま', katakana: 'マ' }, mi: { hiragana: 'み', katakana: 'ミ' },
mu: { hiragana: 'む', katakana: 'ム' }, me: { hiragana: 'め', katakana: 'メ' },
mo: { hiragana: 'も', katakana: 'モ' },

// Y row – only three real cells exist (ya/yu/yo)
ya: { hiragana: 'や', katakana: 'ヤ' }, yu: { hiragana: 'ゆ', katakana: 'ユ' },
yo: { hiragana: 'よ', katakana: 'ヨ' },

// R row
ra: { hiragana: 'ら', katakana: 'ラ' }, ri: { hiragana: 'り', katakana: 'リ' },
ru: { hiragana: 'る', katakana: 'ル' }, re: { hiragana: 'れ', katakana: 'レ' },
ro: { hiragana: 'ろ', katakana: 'ロ' },

// W row – only two real cells; "wo" is the object-particle character を
wa: { hiragana: 'わ', katakana: 'ワ' }, wo: { hiragana: 'を', katakana: 'ヲ' },

// The standalone syllabic n – its own real, single-character sound
n: { hiragana: 'ん', katakana: 'ン' },
};

```

Voiced and Semi-Voiced Sounds

A dakuten mark (゛) or handakuten mark (゜) shifts a base consonant into a real, distinct sound — G/Z/D/B rows, and a P row from H. There's a real, genuine ambiguity worth naming honestly rather than hiding: standard Hepburn romanizes both じ and ぢ as "ji", and both ず and づ as "zu" — a flat romaji-keyed table simply cannot tell these apart. This table maps "ji"/"zu" to their far more common Z-row readings (じ/ず); the rarer ぢ/づ, mostly confined to specific compound-word contexts, aren't reachable through plain romaji input at all.

```
// src/lib/convert.ts — added to the same KANA_MAP object
// G row
ga: { hiragana: 'が', katakana: 'ガ' }, gi: { hiragana: 'ぎ', katakana: 'ギ' },
gu: { hiragana: 'ぐ', katakana: 'グ' }, ge: { hiragana: 'げ', katakana: 'ゲ' },
go: { hiragana: 'ご', katakana: 'ゴ' },

// Z row — "ji"/"zu" deliberately resolve here, per the note above
za: { hiragana: 'ざ', katakana: 'ザ' }, ji: { hiragana: 'じ', katakana: 'ジ' },
zu: { hiragana: 'ず', katakana: 'ズ' }, ze: { hiragana: 'ぜ', katakana: 'ゼ' },
zo: { hiragana: 'ぞ', katakana: 'ゾ' },

// D row — only de/do have their own unambiguous romaji
da: { hiragana: 'だ', katakana: 'タ' }, de: { hiragana: 'で', katakana: 'デ' },
do: { hiragana: 'ど', katakana: 'ト' },

// B row
ba: { hiragana: 'ば', katakana: 'バ' }, bi: { hiragana: 'び', katakana: 'ビ' },
bu: { hiragana: 'ぶ', katakana: 'ブ' }, be: { hiragana: 'べ', katakana: 'ベ' },
bo: { hiragana: 'ぼ', katakana: 'ボ' },

// P row (semi-voiced)
pa: { hiragana: 'ぱ', katakana: 'パ' }, pi: { hiragana: 'ぴ', katakana: 'ピ' },
pu: { hiragana: 'ぷ', katakana: 'プ' }, pe: { hiragana: 'ぺ', katakana: 'ペ' },
po: { hiragana: 'ぽ', katakana: 'ポ' },
```

Combination Sounds (Yōon)

A consonant fused with a small や/ゆ/よ produces a real third category — genuinely three-character romaji tokens like **kya**, **sha**, **cha** — with one more real Hepburn simplification:

the sounds spelled じゃ/じゅ/じょ (from the Z row) romanize as `ja`/`ju`/`jo`, just two characters, not "jya."

```
// src/lib/convert.ts — added to the same KANA_MAP object
kya: { hiragana: 'きゃ', katakana: 'キャ' }, kyu: { hiragana: 'きゅ', katakana: 'キュ'
sha: { hiragana: 'しゃ', katakana: 'シャ' }, shu: { hiragana: 'しゅ', katakana: 'シュ'
cha: { hiragana: 'ちゃ', katakana: 'チャ' }, chu: { hiragana: 'ちゅ', katakana: 'チュ'
nya: { hiragana: 'にゃ', katakana: 'ニャ' }, nyu: { hiragana: 'にゅ', katakana: 'ニュ'
hya: { hiragana: 'ひゃ', katakana: 'ヒャ' }, hyu: { hiragana: 'ひゅ', katakana: 'ヒュ'
mya: { hiragana: 'みゃ', katakana: 'ミャ' }, myu: { hiragana: 'みゅ', katakana: 'ミュ'
rya: { hiragana: 'りゃ', katakana: 'リャ' }, ryu: { hiragana: 'りゅ', katakana: 'リュ'
gya: { hiragana: 'ぎゃ', katakana: 'ギャ' }, gyu: { hiragana: 'ぎゅ', katakana: 'ギュ'
ja: { hiragana: 'じゃ', katakana: 'ジャ' }, ju: { hiragana: 'じゅ', katakana: 'ジュ' },
bya: { hiragana: 'びゃ', katakana: 'ビャ' }, byu: { hiragana: 'びゅ', katakana: 'ビュ'
pya: { hiragana: 'ぴゃ', katakana: 'ピャ' }, pyu: { hiragana: 'ぴゅ', katakana: 'ピュ'
```

DELIBERATELY NOT HANDLED YET

Long vowels (chōon — real double-length vowel sounds) and sokuon (the small つ doubling the following consonant) both need real, dedicated handling of their own — Chapter 3's job specifically. An input containing either isn't converted correctly by anything built in this chapter; that's a scope boundary, not an oversight.

A Naive Two-Character Tokenizer — And a Real Bug

Romaji tokens genuinely vary in length — one character (`a`, `n`), two (`ka`, `shi` ... except `shi` is actually three, which is exactly the trap below), or three (`kya`). A first, reasonable-looking tokenizer only ever tries two characters, then falls back to one:

```
// A naive tokenizer that never tries a 3-character token at all
function convertNaiveV1(romaji: string): string {
  let hiragana = '';
  let i = 0;
  while (i < romaji.length) {
    const two = romaji.slice(i, i + 2);
    const one = romaji.slice(i, i + 1);
    if (KANA_MAP[two]) {
```

```

    hiragana += KANA_MAP[two].hiragana;
    i += 2;
  } else if (KANA_MAP[one]) {
    hiragana += KANA_MAP[one].hiragana;
    i += 1;
  } else {
    hiragana += romaji[i];
    i += 1;
  }
}
return hiragana;
}

```

Run for real against `'kyaku'` — the ordinary word 客 (customer) — this produces `きゃく`, verified directly against real Node output, not just reasoned about in the abstract. Position 0 tries `"ky"` (not a valid token) and then `"k"` alone (also invalid, since only vowels and `n` are genuine one-character tokens), falls all the way through to emitting the raw Latin letter `k`, and only then picks up `"ya"` and `"ku"` correctly.

The base gojūon grid has its own, second, sneakier way to trigger the exact same class of bug — no yōon needed at all. Run against `'sushi'`, the same real function produces `すし`: `"su"` matches correctly, but `"sh"` then `"s"` both fail, the raw letter `s` leaks through, and only then does `"hi"` pick up the remainder — because `shi` itself is one of the base grid's own genuine three-character tokens, this bug was never really about yōon specifically. It's about any token three characters long, wherever it happens to sit in the table.

Adding a Third Length Isn't Enough on Its Own

The obvious fix adds a length-3 check into the same loop — but the order those lengths are tried in matters just as much as trying them at all:

```

// Tries all three lengths — but in ascending order
function convertNaiveV2(romaji: string): string {
  let hiragana = '';
  let i = 0;
  while (i < romaji.length) {
    let matched = false;
    for (let len = 1; len <= 3; len++) {

```

```

const chunk = romaji.slice(i, i + len);
if (KANA_MAP[chunk]) {
  hiragana += KANA_MAP[chunk].hiragana;
  i += len;
  matched = true;
  break;
}
}
if (!matched) {
  hiragana += romaji[i];
  i += 1;
}
}
return hiragana;
}

```

`convertNaiveV2('kyaku')` now actually works — real, verified output `きやく` — since `"k"` and `"ky"` both genuinely fail to match, so the loop reaches length 3 and finds `"kya"`. But `convertNaiveV2('na')` — the single, ordinary syllable `な` — fails in a real, different way, verified producing `んあ` instead of `な`. At position 0, length 1 is tried *first*, and `"n"` is a genuinely valid one-character token on its own (`ん`). The loop matches it immediately and stops, never trying the longer, correct `"na"` at that same position at all.

THE REAL CAUSE: A VALID SHORT TOKEN THAT'S ALSO A PREFIX OF A LONGER ONE

`n` is unusual in this table — it's one of the only single-character tokens that's also a genuine prefix of several longer, different tokens (`na`, `ni`, `nu`, `ne`, `no`). Checking shortest-first means the algorithm can find a real, valid match and stop — before ever discovering a longer, equally real match starting at the exact same position. Trying `"kya"` never hit this problem only because no 1- or 2-character prefix of it happens to be a valid token in its own right.

The Real Fix: Always Check Longest-to-Shortest

Trying lengths in descending order guarantees the longest possible match is always found first — and trying a longer chunk that turns out not to match costs nothing, since the loop simply falls through to a shorter length next:

```
// src/lib/convert.ts – the real, correct tokenizer
const MAX_TOKEN_LENGTH = 3;

export function convert(romaji: string): { hiragana: string; katakana: string } {
  let hiragana = '';
  let katakana = '';
  let i = 0;

  while (i < romaji.length) {
    let matched = false;

    for (let len = MAX_TOKEN_LENGTH; len >= 1; len--) {
      const chunk = romaji.slice(i, i + len).toLowerCase();
      const entry = KANA_MAP[chunk];
      if (entry) {
        hiragana += entry.hiragana;
        katakana += entry.katakana;
        i += len;
        matched = true;
        break;
      }
    }

    if (!matched) {
      hiragana += romaji[i];
      katakana += romaji[i];
      i += 1;
    }
  }

  return { hiragana, katakana };
}
```

Both real bugs are resolved by the identical, single change — checking length 3, then 2, then 1, rather than the other way around. Re-run against every case that broke earlier, this one function correctly produces きやく for "kyaku," すし (not すsひ) for "sushi," and な (not んあ) for "na."

COMPUTING BOTH SCRIPTS IN ONE PASS

`hiragana` and `katakana` are built up together, from the exact same tokenization decisions, in a single loop — not by calling two separate conversion functions. That guarantees the two outputs can never disagree about where one syllable ends and the next begins, since they're never tokenized independently in the first place.

Trying the Engine in Isolation

A quick script — not yet imported into any `.astro` page or component — confirms the engine on real words, run directly with `node` before anything else in this project touches it:

```
console.log(convert('sushi'));
// { hiragana: 'すし', katakana: 'スシ' }

console.log(convert('gakusei'));
// 学生 (student) - { hiragana: 'がくせい', katakana: 'ガクセイ' }

console.log(convert('konnichiwa'));
// { hiragana: 'こんにちは', katakana: 'コンニチワ' }
```

That last one is worth pausing on. The real, formal greeting is written こんにちは — using the character は (normally "ha"), pronounced "wa" specifically in this one fossilized grammatical role. Typed literally as "konnichiwa," this engine — correctly, given only what it knows so far — produces こんにちは instead, a genuinely common casual spelling that plenty of native speakers use interchangeably, but not the traditionally "correct" one. Recognizing when a trailing "wa" really means は is exactly the kind of particle exception Chapter 3 builds real handling for.

Where This Engine Will Actually Live

Nothing built in this chapter depends on Astro at all — `KANA_MAP` and `convert()` are plain TypeScript, sitting in `src/lib/convert.ts` exactly where Chapter 1's own project structure already reserved a place for them, testable with nothing more than `node` or a browser console. Chapter 4 imports this exact, unmodified module directly into the real input/output UI; Chapter 5 is where it gets a genuine server-side alternative to be measured against, rather than being assumed to belong on one side or the other.

Hands-On Exercises

EXERCISE 1

Build the complete KANA_MAP (vowels through yōon) and the final, correct `convert()` function in `src/lib/convert.ts`, then run it against "sushi", "kyaku", and "gakusei" (学生, student), confirming correct hiragana and katakana output for all three against real node output.

 [View solution](#)

EXERCISE 2

Build `convertNaiveV1` (the fixed 2-then-1-character tokenizer) exactly as this chapter describes. Run it against "kyaku" and confirm the real "kゃく" output, then run it against "kaki" (a word with no 3-character tokens anywhere in it) and confirm it produces the correct result there — showing the bug specifically depends on a 3-character token being present at that position, not a universal failure.

 [View solution](#)

EXERCISE 3

Build `convertNaiveV2` (checking lengths 1, 2, 3 in that order) and confirm it produces the correct result for "kyaku" but the wrong result ("んあ") for "na". Then explain in your own words why "n" specifically is the token that exposes this bug, when most other single-character or short tokens wouldn't.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **KANA_MAP** — the real gojūon grid, plus voiced/semi-voiced rows and yōon combinations, keyed by romaji, in `src/lib/convert.ts`
- **Known limitation** — "ji"/"zu" can't distinguish じ/ぢ or ず/づ from romaji alone; this table resolves to the more common reading
- **Deliberately deferred** — long vowels and sokuon (っ) both need dedicated handling, built in Chapter 3

- **Bug 1** — a tokenizer that never tries 3 characters breaks on any 3-character token, yōon or not — verified on both "kyaku" (くやく) and the base-grid case "sushi" (すし)
- **Bug 2** — even trying all three lengths, checking shortest-first can match a valid short token ("n") before ever considering a longer, correct one ("na")
- **The fix** — always check lengths longest-to-shortest; a failed longer match costs nothing and simply falls through
- **Still framework-agnostic** — this module has no Astro dependency yet, by design, keeping Chapter 5's own client-vs-server comparison genuinely open
- **Next chapter:** Handling Real Edge Cases — long vowels, sokuon, and particle exceptions like は → "wa"

CHAPTER 3: HANDLING REAL EDGE CASES: LONG VOWELS, SOKUON & PARTICLE EXCEPTIONS

Romaji to Kana Converter: Astro

Chapter 3 · Handling Real Edge Cases: Long Vowels, Sokuon (っ) & Particle Exceptions

Chapter 2's own closing warn-box named two things deliberately left unhandled — long vowels and the small つ — and its very last test surfaced a third: `convert('konnichiwa')` produced こんにちは instead of the traditionally correct こんにちは. This chapter builds real, tested handling for all three, adding directly onto the same `src/lib/convert.ts` module Chapter 2 started. Every example below was independently re-run through a real Node script before being written down — two of them exposed genuine bugs on the first attempt, caught the exact same way Chapter 2's own tokenizer bugs were: by actually running the code rather than trusting it looked right.

Long Vowels: How Chōon Is Actually Spelled

A long (double-length) vowel sound is real and phonemically distinct in Japanese — おばさん (obasan, aunt) and おばあさん (obaasan, grandmother) are genuinely different words, not the same word said carefully. But hiragana and katakana don't spell a long vowel the same way, and hiragana itself doesn't even spell it the same way for every vowel:

VOWEL ROW	REAL HIRAGANA CONVENTION	REAL EXCEPTIONS
あ (a)	Extend with あ — おかあさん (okaasan, mother)	—
い (i)	Extend with い — おにいさん (oniisan, older brother)	—
う (u)	Extend with う — くうき (kuuki, air)	—
え (e)	Usually extend with い — せんせい (sensei, teacher)	Sometimes え — おねえさん (oneesan, older sister)
お (o)	Usually extend with う — とうきょう (toukyou)	Sometimes お — とい (tooi, far), おおきい (ookii, big)

Katakana avoids this split entirely — a long vowel in katakana is always written with a single mark, ー (chōonpu), regardless of which vowel is being extended: コーヒー (kōhī, coffee), ラーメン (rāmen, ramen). That's the real, standard convention katakana (used overwhelmingly for loanwords) actually follows, not a simplification invented for this course.

A REAL, UNRESOLVABLE AMBIGUITY

Hepburn romanization with macrons (ā/ī/ū/ē/ō) collapses the え/い and お/う distinction on purpose — a macron just says "this vowel is long," not which of the two real hiragana spellings applies. Knowing whether a macron ō came from とう or とお requires knowing the specific word; it can't be recovered from the romaji alone. This converter makes a documented default choice below, and that choice will be wrong for real, correctly-spelled exception words like とおい and おおきい.

A First Attempt — And a Real Bug

The plan: expand each macron character into an ASCII digraph before tokenizing, so the longest-match loop already built in Chapter 2 can do the rest. Hiragana gets the "best guess" native-word extension (え→い, お→う); katakana gets the same vowel doubled, since that's what will need collapsing into a single ー:

```
// src/lib/convert.ts
const MACRON_TO_HIRAGANA: Record<string, string> = {
  ā: 'aa', ī: 'ii', ū: 'uu',
  ē: 'ei', // best-guess default — real words vary, see the warn-box above
  ō: 'ou', // best-guess default — real words vary, see the warn-box above
};

const MACRON_TO_KATAKANA: Record<string, string> = {
  ā: 'aa', ī: 'ii', ū: 'uu', ē: 'ee', ō: 'oo',
};

function expandMacrons(romaji: string, table: Record<string, string>): string {
  let out = '';
  for (const ch of romaji) out += table[ch.toLowerCase()] ?? ch;
  return out;
}
```

Then a first, plausible pass at turning a doubled katakana vowel into `ー` — scan the already-tokenized string, and whenever two consecutive characters happen to share the same trailing vowel sound, collapse the second into `ー`:

```
// A post-processing pass over the ALREADY-BUILT katakana string
function collapseChoonpuNaive(katakana: string, vowelOf: Record<string, string>): string {
  let out = '';
  let prevVowel: string | null = null;
  for (const ch of katakana) {
    const vowel = vowelOf[ch];
    if (vowel && vowel === prevVowel) {
      out += 'ー';
    } else {
      out += ch;
    }
    prevVowel = vowel ?? null;
  }
  return out;
}
```

Run for real against the katakana for `"raamen"`, this correctly collapses `ラーメン` into `ラームン`. Run the identical function against the katakana for the ordinary word `"ikimasu"` (行きます, I go) — a word with no long vowel in it at all — and it produces `イーマス` instead of the correct `イキマス`, verified directly against real Node output rather than assumed from the algorithm's own description.

THE REAL CAUSE: COMPARING CHARACTERS, NOT SYLLABLES

イ (from `i`) and キ (from `ki`) are two entirely different, unrelated syllables that simply happen to both end in an "i" sound — the same coincidence that makes "kit" and "sit" rhyme in English without either word containing a doubled letter. `collapseChoonpuNaive` only looks at the two most recently emitted *characters*, with no memory of which whole *token* either one came from — so two unrelated syllables that happen to share a trailing vowel get merged exactly as if a genuine vowel had been typed twice in a row. A real vowel-lengthening collapse has to compare against the vowel of the last *matched token*, not the last accumulated character.

The Real Fix: Track the Vowel Inside the Tokenizer Itself

Rather than patching the finished string afterward, the chōonpu decision moves inside the same loop that's already matching tokens — it always knows exactly which real syllable it just emitted, so it can never confuse "ends in the same vowel sound" with "is a genuine repeated vowel":

```
// src/lib/convert.ts — replaces the naive post-processing pass entirely
function tokenizeScript(romaji: string, useChoonpu: boolean): string {
  let out = '';
  let i = 0;
  let lastVowel: string | null = null;

  while (i < romaji.length) {
    let matched = false;

    for (let len = MAX_TOKEN_LENGTH; len >= 1; len--) {
      const chunk = romaji.slice(i, i + len).toLowerCase();
      const entry = KANA_MAP[chunk];
      if (entry) {
        // chunk.length, NOT the loop variable len — see the warn-box below
        const isBareVowel = chunk.length === 1 && 'aiueo'.includes(chunk);

        if (useChoonpu && isBareVowel && chunk === lastVowel) {
          out += '—';
        } else {
          out += useChoonpu ? entry.katakana : entry.hiragana;
        }

        lastVowel = chunk === 'n' ? null : chunk[chunk.length - 1];
        i += len;
        matched = true;
        break;
      }
    }

    if (!matched) {
      out += romaji[i];
      lastVowel = null;
      i += 1;
    }
  }
}
```

```
return out;
}
```

A SECOND REAL BUG, FOUND TESTING "KÔHĪ"

A first version of this function checked `len === 1` instead of `chunk.length === 1` to decide whether a match was a bare vowel. Near the end of a string, `romaji.slice(i, i + 3)` can genuinely return fewer than 3 characters — there's nothing left to slice — so a length-3 attempt at the very last character of `"koohii"` returns the 1-character string `"i"`, which correctly matches the vowel token, but with `len` still equal to 3. Checking the loop variable instead of the matched string's own real length made the function think this was a 3-character match, never recognizing it as the bare vowel it actually was — re-run for real, that buggy version produces `コーヒイ` instead of the correct `コーヒー`. Switching the check to `chunk.length` — the true length of what was actually found, regardless of which loop iteration found it — fixed it, re-verified producing the correct `コーヒー`.

Both scripts now need their own full pass, expanded with their own macron table, because they genuinely diverge on exactly the case this chapter is about:

```
console.log(tokenizeScript(expandMacrons('kôhî', MACRON_TO_HIRAGANA), false));
// こうひい
console.log(tokenizeScript(expandMacrons('kôhî', MACRON_TO_KATAKANA), true));
// コーヒー

console.log(tokenizeScript(expandMacrons('tôkyô', MACRON_TO_HIRAGANA), false));
// とうきょう – the real, correct spelling
console.log(tokenizeScript(expandMacrons('tôi', MACRON_TO_HIRAGANA), false));
// とうい – the real word is spelled とうい; this is the honest limitation from the warn
```

CHAPTER 2'S "ONE SHARED LOOP" PROMISE DOESN'T SURVIVE LONG VOWELS

Chapter 2's own tip-box promised hiragana and katakana are always built "from the exact same tokenization decisions, in a single loop." Long vowels break that promise on purpose: `とうきょう` correctly needs a `ょう` in hiragana but — in katakana at the identical position, so the two scripts now genuinely need two separate macron-expanded inputs and two separate tokenizer passes. The single-pass guarantee still holds for everything *without* a long vowel in it — which is most input — but it's no longer universal.

Sokuon: The Small つ

A doubled consonant in romaji (other than a doubled `n`) signals a small つ/ツ immediately before the syllable that follows — `きって` (kitte, stamp), `がっこう` (gakkou, school). The small tsu itself carries no sound of its own; it's a brief, silent held pause, then the following consonant releases normally:

```
// src/lib/convert.ts
const SMALL_TSU = { hiragana: 'っ', katakana: 'ツ' };

// Returns how many characters the sokuon trigger itself consumes (0 if none) -
// the actual syllable that follows is left for the normal tokenizer loop.
function sokuonLength(romaji: string, i: number): number {
  const c = romaji[i];
  if (!c || 'aiueo'.includes(c)) return 0;

  // Real Hepburn irregularity: gemination before chi/cha/chu/cho is spelled
  // with a "t", not a doubled "c" - まっちゃ is "matcha," never "maccha."
  if (c === 't' && romaji.slice(i + 1, i + 3) === 'ch') return 1;

  // Doubled n is NOT sokuon - see こんにちは below.
  if (c !== 'n' && romaji[i + 1] === c) return 1;

  return 0;
}
```

Wired into `tokenizeScript`, this is checked before every ordinary token attempt: if a sokuon trigger is found, emit the small tsu, advance past just the trigger, and let the loop pick up the real syllable from there on its very next iteration.

```
console.log(tokenizeScript('kitte', false)); // きって
console.log(tokenizeScript('gakkou', false)); // がっこう
console.log(tokenizeScript('matcha', false)); // まっちゃ - the real "tch" exception, ve
console.log(tokenizeScript('kekkon', false)); // けっこん (結婚, marriage)
```

WHY "MATCHA" ISN'T "MACCHA"

Trace it by hand: at `t`, the next two characters are exactly `"ch"`, so the special case fires and consumes only the `t` — emitting `っ` and leaving the string positioned right at `"cha"`, which the ordinary tokenizer then matches as a normal 3-character token, `ちゃ`. The doubled-consonant rule alone (`t` followed by another `t`) would never have fired here at all, since the second letter is `c`, not `t` — this is a genuinely separate, irregular case, not a variation on the ordinary rule.

The doubled-`n` exclusion matters for a real, common word — `こんにちは` (konnichiwa, hello) is ko-n-ni-chi-wa, five real syllables with no small tsu anywhere in it. Treating a doubled `n` as sokuon would insert a phantom `っ` that doesn't belong:

```
console.log(tokenizeScript('konnichiwa', false));
// こんにちはわ – correct so far as sokuon goes (no phantom っ), but still ends in
// わ rather than は – that's a completely different, particle-level problem, next
```

The Real は/へ/を Particle Exceptions

Three hiragana keep an old, "wrong-looking" spelling in one specific grammatical role — the character is written one way but pronounced another:

CHARACTER	ORDINARY READING	AS A PARTICLE	REAL EXAMPLE
は	ha	wa (topic marker)	これは本です — kore wa hon desu
へ	he	e (direction marker)	これへ行きます — kore e ikimasu
を	wo	o (object marker)	これをください — kore o kudasai

This converter runs romaji *into* kana, so the direction that actually matters is: given typed `"wa"`, `"e"`, or `"o"`, when should the output be は/へ/を instead of the ordinary わ/え/お? A workable, honest heuristic: only when that romaji appears as its own complete, space-separated word — never when it's a letter sequence inside a longer word:

```
// src/lib/convert.ts
const PARTICLE_OVERRIDES: Record<string, string> = { wa: 'ha', e: 'he', o: 'wo' };

export function convertSentence(romaji: string): { hiragana: string; katakana: string } {
  const words = romaji.split(' ');
```

```

const hiraganaParts: string[] = [];
const katakanaParts: string[] = [];

for (const word of words) {
  const lower = word.toLowerCase();
  const effective = PARTICLE_OVERRIDES[lower] ?? word;
  hiraganaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_HIRAGANA), false));
  katakanaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_KATAKANA), true));
}

return { hiragana: hiraganaParts.join(' '), katakana: katakanaParts.join(' ') };
}

```

This resolves both grammatically unambiguous cases correctly, and correctly leaves わたし (watashi, I) — which genuinely contains the letters `"wa"` but never as its own word — completely untouched:

```

console.log(convertSentence('kore wa hon desu').hiragana);
// これはほんです – the topic particle, correctly は

console.log(convertSentence('kore o kudasai').hiragana);
// これをください – the object particle, correctly を

console.log(convertSentence('watashi wa gakusei desu').hiragana);
// わたしはがくせいです – the "wa" inside watashi is left alone; only the
// standalone word is overridden

```

WHERE THIS HEURISTIC GENUINELY BREAKS: A REAL, UNRESOLVABLE AMBIGUITY

`"wa"` and `"o"` standing alone are almost always the particle — there's little else a lone `"wa"` or `"o"` would normally be. But standalone `"e"` is a real, common Japanese word in its own right — 絵 (e), meaning "picture." Compare two genuinely grammatical sentences:

- `"kore e ikimasu"` ("I'm going this way") — `"e"` really is the へ particle
- `"kono e wa kirei desu"` ("This picture is pretty") — `"e"` here means 絵, and should stay え, not へ

A word-boundary rule alone can't tell these two sentences apart — both have a standalone `"e"` in the identical grammatical position. This converter always resolves standalone `"e"` to へ, which is right for the first sentence and genuinely wrong for the second. Correctly distinguishing them would

need real part-of-speech context — something well outside a lookup-table-driven converter — so this is documented as a known, permanent limitation rather than quietly hidden.

Verified against both real sentences directly — the second one confirms the honest limitation actually exists, not just that it's theoretically possible:

```
console.log(convertSentence('kore e ikimasu').hiragana);
// これ へ いきます - correct

console.log(convertSentence('kono e wa kirei desu').hiragana);
// この へ は きれい です - "wa" correctly resolves to は, but "e" incorrectly
// becomes へ instead of the correct え (this sentence is really about a picture)
```

こんにちは: A Whole-Word Exception, Not a Grammatical One

こんにちは itself needs a genuinely different fix from the general particle rule above. Its own は isn't a standalone particle word at all — it's fused inside one continuous greeting, historically built from "today, as for..." with the rest of the sentence traditionally left unsaid. The word-boundary heuristic can't reach it, because there's no space anywhere near that は to detect. This needs a small, explicit dictionary of whole-word exceptions, checked before anything else runs:

```
// src/lib/convert.ts
const LEXICALIZED_EXCEPTIONS: Record<string, { hiragana: string; katakana: string }> =
  konnichiwa: { hiragana: 'こんにちは', katakana: 'コンニチハ' },
  konbanwa: { hiragana: 'こんばんは', katakana: 'コンバンハ' },
};
```

TWO GENUINELY DIFFERENT KINDS OF EXCEPTION

This module now has two structurally different fixes living side by side: the general

`PARTICLE_OVERRIDES` rule, which applies whenever a real grammatical pattern holds (a standalone word between two others), and `LEXICALIZED_EXCEPTIONS`, a small closed list of specific, memorized whole words whose irregular spelling can't be derived from any general rule at all. `こんばんは` (konbanwa, good evening) is a real, second example of the exact same lexicalized pattern — worth

knowing the category exists, since a third such word could show up later and need the same treatment.

Putting It All Together

The final `convert()` checks the lexicalized exception dictionary first, applies the particle override to a standalone word second, then runs the full macron-expansion-plus-sokuon-aware tokenizer for each script — every layer from this chapter, combined:

```
// src/lib/convert.ts — the real, exported Chapter 3 convert()
export function convert(romaji: string): { hiragana: string; katakana: string } {
  const words = romaji.split(' ');
  const hiraganaParts: string[] = [];
  const katakanaParts: string[] = [];

  for (const word of words) {
    const lower = word.toLowerCase();

    const lexical = LEXICALIZED_EXCEPTIONS[lower];
    if (lexical) {
      hiraganaParts.push(lexical.hiragana);
      katakanaParts.push(lexical.katakana);
      continue;
    }

    const effective = PARTICLE_OVERRIDES[lower] ?? word;
    hiraganaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_HIRAGANA), false));
    katakanaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_KATAKANA), true));
  }

  return { hiragana: hiraganaParts.join(' '), katakana: katakanaParts.join(' ') };
}
```

Every feature from this chapter, exercised together in one sentence — a doubled consonant and a direction particle at once — plus the exact greeting Chapter 2 ended on:

```
console.log(convert('gakkou e ikimasu'));
// { hiragana: 'がっこう え いきます', katakana: 'ガッコウ ヘ イキマス' }

console.log(convert('konnichiwa'));
// { hiragana: 'こんにちは', katakana: 'コンニチハ' } – finally correct, resolving
// the exact cliffhanger Chapter 2 ended on
```

Nothing built across this chapter or the last one depends on Astro yet — [src/lib/convert.ts](#) is still plain TypeScript. Chapter 4 imports this exact, unmodified module into a real input/output UI; Chapter 5 is where it finally gets a server-side alternative to be measured against.

Hands-On Exercises

EXERCISE 1

Build `sokuonLength` exactly as this chapter describes, wire it into a tokenizer, and verify it against three real words: "zutto" (ずっと, all along), "issho" (いっしょ, together), and "matcha" (matcha). Then explain, in your own words, why checking for "ch" specifically after a "t" has to happen before the general doubled-consonant check runs — what would go wrong if the order were reversed?

 [View solution](#)

EXERCISE 2

Build `collapseChoonpuNaive` (the character-comparison version) and confirm it works for "raamen" but produces "イーマス" for "ikimasu". Then build the fixed `tokenizeScript` version and confirm it correctly produces "イキマス" for the same word while still correctly producing "ラーメン" for "raamen".

 [View solution](#)

EXERCISE 3

Build `convertSentence` with the particle overrides, then run it against both "kore e ikimasu" and "kono e wa kirei desu". Confirm the first is correct and the second is wrong (produces え where え would be

correct), and write one sentence explaining why no purely lookup-table-based fix could resolve this specific ambiguity.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **Long vowels** — hiragana extends あ/い/う-row vowels with themselves, but usually extends え-row with い and お-row with う (real exceptions exist: とおい, おおきい); katakana always uses one mark, —
- **Bug 1** — collapsing a long vowel by comparing accumulated katakana characters (not matched tokens) wrongly merges unrelated syllables that happen to share a trailing vowel (イキ → イー)
- **Bug 2** — checking the loop variable `len` instead of `chunk.length` misses a bare-vowel match truncated by the string's own end
- **Sokuon** — a doubled consonant (except doubled `n`) inserts つ/ッ before the following syllable; gemination before ち/ちや/ちゅ/ちょ is spelled with "t" ("matcha," never "maccha")
- **Particle exceptions** — standalone "wa"/"e"/"o" override to は/へ/を; standalone "e" is genuinely ambiguous with 絵 (picture) and has no lookup-table fix
- **Lexicalized exceptions** — こんにちは and こんにちは need a small whole-word dictionary, since their own は isn't a standalone particle at all
- **Still framework-agnostic** — `src/lib/convert.ts` is complete and correct-as-documented, with no Astro dependency anywhere yet
- **Next chapter:** Building the Input/Output UI in Astro — this module gets its first real caller

CHAPTER 4: BUILDING THE INPUT/OUTPUT UI IN ASTRO

Romaji to Kana Converter: Astro

Chapter 4 · Building the Input/Output UI in Astro

Chapters 2 and 3 built a complete, correctly-tested conversion engine with zero Astro dependency anywhere in it. This chapter gives `src/lib/convert.ts` its first real caller — replacing Chapter 1's own placeholder page with an actual input box and live kana output, built entirely with a plain client-side script and no framework, no island, and no extra dependency installed.

A Plain `<script>` Tag Is All This Needs

Astro processes an ordinary `<script>` tag — one with no attributes beyond an optional `src` — through a real, specific pipeline, verified directly against Astro's own current documentation rather than assumed:

- **It's bundled by Vite.** The script becomes a real ES module, deduplicated automatically if the same component ever renders more than once on a page, and small scripts get inlined into the HTML automatically.
- **It supports real ES module imports.** A default script can import both local files from `src/` and npm packages — which is exactly what lets this page pull in `convert` from `src/lib/convert.ts` with an ordinary relative import.
- **It's TypeScript by default.** Astro's own documentation states this plainly — no extra config, no `lang="ts"` attribute, nothing to opt into.

The one thing that turns all of this off is the `is:inline` directive — Astro's own documentation is direct about what it does: a script marked `is:inline` is rendered into the HTML exactly as written, with no TypeScript support and no import resolution at all. That's the right choice for an external CDN script or something already sitting in `public/`; it's the wrong choice here, since this page's whole script depends on importing a real local TypeScript module.

The Page Markup

A text input, and two separate output areas — one per script, matching how `convert()` has returned both since Chapter 2:

```
<!-- src/pages/index.astro -->
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Romaji to Kana Converter</title>
  </head>
  <body>
    <main>
      <h1>Romaji to Kana Converter</h1>

      <label for="romaji-input">Type romaji:</label>
      <input id="romaji-input" type="text" placeholder="e.g. konnichiwa" autocomplete="off">

      <section>
        <h2>Hiragana</h2>
        <p id="hiragana-output"></p>
      </section>

      <section>
        <h2>Katakana</h2>
        <p id="katakana-output"></p>
      </section>
    </main>

    <!-- script goes here — see the next section -->
  </body>
</html>
```

No Astro frontmatter fence is needed anywhere in this file — nothing about this page runs at build time or request time on the server at all. Every part of what makes this page actually work happens in the browser, after the static HTML above has already loaded.

Wiring the Script

Imported directly, called on every keystroke, with the two output elements updated from the exact same `{ hiragana, katakana }` shape `convert()` has returned since Chapter 2:

```
<script>
  import { convert } from '../lib/convert';

  const input = document.getElementById('romaji-input') as HTMLInputElement;
  const hiraganaOutput = document.getElementById('hiragana-output')!;
  const katakanaOutput = document.getElementById('katakana-output')!;

  input.addEventListener('input', () => {
    const { hiragana, katakana } = convert(input.value);
    hiraganaOutput.textContent = hiragana;
    katakanaOutput.textContent = katakana;
  });
</script>
```

Running `npm run dev` and typing into the input box updates both output areas on every single keystroke — no submit button, no page reload, and (per the section above) no bundler configuration of any kind beyond what a default Astro scaffold already provides.

A Real, Honest Finding: Converting Live, Mid-Token

Converting on every keystroke means the engine is regularly asked to convert a string that isn't a complete word yet — and Chapter 2's own longest-to-shortest tokenizer has a real, predictable answer for that: whatever hasn't matched a real token yet passes through untouched. Typing the word `kya` (客-adjacent syllable きゃ) one letter at a time, and calling `convert()` after each one, produces exactly this, verified directly against real output:

```
after typing "k":  "k"
after typing "ky": "ky"
after typing "kya": "きゃ"
```

EXPECTED BEHAVIOR, NOT A BUG — BUT WORTH KNOWING ABOUT

While a three-character token like `kya` is still incomplete, the tokenizer correctly has no valid match for `"k"` or `"ky"` on their own, so both letters pass straight through as raw Latin text — exactly the

same "no match, emit the raw character" fallback Chapter 2 built for genuinely unrecognized input. The instant the third letter completes a real token, the output snaps directly from raw Latin to the correct kana in one keystroke. This is the algorithm working exactly as specified, not a glitch — but a first-time user watching text flicker between "unconverted" and "converted" while typing a yōon syllable could easily read it as broken if this behavior isn't expected going in.

`こんにちは` shows the same effect on a larger scale: since `convert()` checks the lexicalized-exception dictionary against the *whole* word, typing `"konnichiw"` — one letter short of the full greeting — produces the ordinary tokenized-but-uncorrected output, and only the final `"a"` completes the exact string the dictionary is keyed on, snapping the whole output to `こんにちは` in a single keystroke.

NO DEBOUNCING NEEDED — AND THAT'S WORTH NOTICING

Converting on every keystroke, with no delay, works fine here specifically because `convert()` is a plain, synchronous, entirely local function call — there's no network request to rate-limit, and no latency for a fast typist to outrun. A server-backed version of this exact same feature would have a real reason to debounce every keystroke into a single delayed request instead of firing one per letter — that's precisely the kind of tradeoff Chapter 5's own client-vs-server comparison exists to measure directly, rather than this chapter simply declaring one approach better.

Trying It End to End

With the script wired in, typing each of the words already verified across Chapters 2 and 3 reproduces the exact same output live in the browser that `node` already confirmed on the command line:

TYPED	HIRAGANA OUTPUT	KATAKANA OUTPUT
<code>sushi</code>	すし	スシ
<code>gakusei</code>	がくせい	ガクセイ
<code>konnichiwa</code>	こんにちは	コンニチハ

Nothing about this UI depends on where the conversion logic actually runs — `convert()` is called the same way whether it's the local function this chapter imports directly, or (per Chapter 5) a fetch call to a real endpoint returning the identical shape. That's deliberate: this chapter

builds the UI around a stable interface, so the comparison in Chapter 5 can swap out what's behind it without touching the markup or the event listener at all.

Hands-On Exercises

EXERCISE 1

Build `index.astro` exactly as this chapter describes, run `npm run dev`, and type "kya" into the input box one letter at a time. Confirm you see the same live intermediate output this chapter verified (k, then ky, then きゃ) before writing down what you observed.

 [View solution](#)

EXERCISE 2

Add a temporary `console.log` inside the input event handler, logging the raw input value and the converted hiragana/katakana on every keystroke. Open the browser console and confirm it fires exactly once per keystroke — not more, not less — while typing a full word.

 [View solution](#)

EXERCISE 3

In your own words, explain why this page's on-input conversion needs no debouncing at all, and name specifically what would have to change about where the conversion logic runs for debouncing to genuinely become necessary.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **index.astro** — a plain input box plus two output areas, no frontmatter needed since nothing runs server-side here
- **A default `<script>` tag** — bundled by Vite, TypeScript by default, real ES module imports supported, verified against Astro's own current documentation

- **is:inline** — the one thing that turns all of that off; the wrong choice here, since this page depends on importing a real local module
- **Live conversion, verified** — an incomplete multi-character token (k, ky) passes through as raw Latin text until the full token completes, then snaps to the correct kana in one keystroke — expected, not a bug
- **No debouncing needed** — `convert()` is synchronous and entirely local, with no network request to rate-limit
- **Stable interface** — the UI calls `convert()` the same way regardless of where the logic behind it actually runs, setting up Chapter 5's own comparison cleanly
- **Next chapter:** Client-Side vs. Server-Side Conversion — building a real server-rendered alternative and measuring it directly against this one

CHAPTER 5: CLIENT-SIDE VS. SERVER-SIDE CONVERSION: WHERE SHOULD THE LOGIC ACTUALLY RUN?

Romaji to Kana Converter: Astro

Chapter 5 · Client-Side vs. Server-Side Conversion: Where Should the Logic Actually Run?

Every chapter since the project began has left this question genuinely open — Chapter 1's own comparison table named it directly, and Chapter 4 closed with a tip-box promising this chapter would measure it rather than argue about it. This chapter does exactly that: it builds a real, working server-side alternative to Chapter 4's own client-side conversion, then benchmarks the two directly with real code rather than assuming which one is faster.

Building the Server-Side Alternative

Chapter 1 left this project in Astro's own default `output: 'static'` mode, deliberately, with no adapter installed — nothing in the project needed on-demand rendering yet. This chapter is the first genuine exception: a real API route that runs `convert()` on the server needs to execute per request, not once at build time, which means it needs an adapter after all — verified back in Chapter 1 against Astro's own documentation: on-demand rendering requires one, full stop.

```
# Add the Node adapter — needed for exactly one route from here on
npx astro add node
```

The rest of the project stays static by default. Adding an adapter doesn't switch the whole site over to `output: 'server'` — it just makes on-demand rendering *available*, opt-in per route via that route's own `export const prerender = false`:

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import node from '@astrojs/node';

export default defineConfig({
  adapter: node({ mode: 'standalone' }),
  // output left unset — still 'static' by default. Only the one route
```

```
// below opts into on-demand rendering; everything else is unaffected.
});
```

The route itself imports the exact same, unmodified `convert` function Chapters 2 and 3 already built and tested — no second copy of the algorithm, no drift between the two versions:

```
// src/pages/api/convert.ts
import type { APIRoute } from 'astro';
import { convert } from '../../lib/convert';

export const prerender = false;

export const POST: APIRoute = async ({ request }) => {
  const { romaji } = await request.json();
  const result = convert(romaji ?? '');
  return new Response(JSON.stringify(result), {
    status: 200,
    headers: { 'Content-Type': 'application/json' },
  });
};
```

THIS IS THE EXACT MOMENT CHAPTER 1'S OWN EXERCISE 3 WAS POINTING TOWARD

Chapter 1's own third exercise had the reader add `prerender = false` to a scratch page with no adapter installed, specifically to trigger a real build failure and feel the requirement directly. This chapter is the real, non-scratch version of that exact situation — the first genuine route in the whole project that actually needs what that exercise was rehearsing for.

A Second Way to Call It

Chapter 4's own input box still calls `convert()` directly, unchanged. A second box, added to the same page, calls the new endpoint instead — same shape in, same shape out, but through a real network request this time:

```
<script>
  import { convert } from '../../lib/convert';
```

```
// ...Chapter 4's own client-side box, unchanged above this line...

const serverInput = document.getElementById('server-input') as HTMLInputElement;
const serverHiragana = document.getElementById('server-hiragana-output')!;
const serverKatakana = document.getElementById('server-katakana-output')!;

serverInput.addEventListener('input', async () => {
  const res = await fetch('/api/convert', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ romaji: serverInput.value }),
  });
  const { hiragana, katakana } = await res.json();
  serverHiragana.textContent = hiragana;
  serverKatakana.textContent = katakana;
});
</script>
```

Typing the same words into both boxes produces identical output — `sushi`, `kyaku`, and `konnichiwa` all convert exactly the same way whether the request stays in the browser or makes a real round trip to the server, because both boxes ultimately call the exact same `convert()` function underneath. The only real difference between them is what it costs to get there — measured directly below, not assumed.

Measuring the Real Difference

Rather than guessing, a small, real benchmark compares a direct, in-process call to `convert()` against a real HTTP round trip carrying the identical JSON request/response shape this chapter's own `/api/convert` route uses — a real local Node server and a real `fetch` client, run for real rather than described in the abstract:

```
// Direct, in-process call — 2,000 runs of convert('konnichiwa')
for (let i = 0; i < 2000; i++) convert('konnichiwa');
// total: 2.72ms → 1.360µs per call

// Real HTTP round trip to a local server exposing the identical
// POST /api/convert contract, 2,000 sequential requests
for (let i = 0; i < 2000; i++) {
```

```
const res = await fetch('http://127.0.0.1:4321/api/convert', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ romaji: 'konnichiwa' }),
});
await res.json();
}
// total: 28,168.47ms → 14,084.236µs per call
```

APPROACH	AVERAGE TIME PER CALL	2,000-CALL TOTAL
Direct, in-process (Chapter 4's approach)	1.36 microseconds	2.72 ms
HTTP round trip to a local server	14,084 microseconds (~14ms)	28,168 ms

That's a real, measured **~10,353×** difference — re-run at a smaller sample size (500 calls) to check it wasn't a fluke, the per-call HTTP latency held steady at roughly the same ~14ms, with the ratio landing at ~7,018× that time. The exact multiple moves around a little run to run, since the direct call is fast enough to be sensitive to normal timer noise — but the real, load-bearing number, the ~14ms per HTTP round trip, was consistent across both runs.

A REAL METHODOLOGY NOTE

This benchmark uses a small, standalone Node HTTP server exposing the identical JSON request/response contract this chapter's own Astro endpoint uses, rather than a literal browser session against a running Astro dev server — that kind of end-to-end browser measurement isn't something this written format can run directly. What's measured here is real, though: an actual HTTP round trip (connection handling, JSON serialization on both ends, real event-loop scheduling) against an actual server process, on the same machine, with zero real network latency in the mix at all. A production deployment (Chapter 7) would add genuine network latency on top of this local baseline, not remove any of it.

"LOCALHOST IS BASICALLY FREE" DOESN'T HOLD UP HERE

A same-machine HTTP round trip is often assumed to be nearly instantaneous, with "the network" being the real cost that only shows up once a request actually leaves the machine. The measured numbers here say otherwise: even with zero physical network hop involved, a real HTTP request still pays for a connection, two JSON serialization/parse steps, and several real trips through the event loop — costs a direct function call never pays at all, since it never leaves the current call stack.

What This Means for Chapter 4's Own Debouncing Question

Chapter 4's own tip-box named the exact condition under which debouncing would become worth doing: moving `convert()` behind a real network call. That condition is now true for the server-side box built in this chapter — converting on every keystroke through `/api/convert` means paying something in the neighborhood of this chapter's own measured ~14ms, per letter typed, for a feature that costs approximately 1.36 microseconds when kept client-side. A debounce delay on the server-side box specifically would be a genuinely reasonable addition; Chapter 4's own client-side box still needs none, for the exact reason it never did.

So Which Should This App Actually Use?

Given the real evidence gathered in this chapter, the honest answer for this specific app isn't "it depends" — it's a real, measured recommendation: **Chapter 4's client-side approach is the objectively better choice for this feature**, and the server-side route built in this chapter is worth having built and understood, not worth actually shipping as this app's own default.

- **No real benefit from moving it server-side.** `convert()` touches no secret, no shared state, and no data any user shouldn't already have direct access to — none of the usual reasons a feature genuinely needs to run on a server apply here.
- **A real, measured cost for doing it anyway.** ~10,000× slower per call isn't a rounding error — it's the kind of number that would show up as real, felt lag on a slower connection or a busier server, for a feature that's otherwise instant.
- **Works with no server running at all.** The client-side version keeps working the instant JavaScript loads, with no dependency on this project's own backend being reachable, deployed, or even online.

This is a genuinely different verdict from this course's own Angular & Express sibling, which commits to server-side conversion from its own first chapter — worth naming directly, since that course's own reason was structural (giving Angular a real backend it doesn't otherwise have), not a claim that server-side conversion is generally better. This chapter reaches the opposite conclusion for a completely different, and for this specific app more directly relevant, reason: real, measured evidence that this particular feature has nothing to gain from a server and a real, measured cost for adding one anyway.

THE SERVER ROUTE ISN'T WASTED WORK

Building `/api/convert` wasn't pointless even though it loses this comparison — it's the concrete proof that Chapter 1's own "no database needed" architectural reasoning holds up under real measurement, not just under a plausible-sounding argument. A future feature genuinely requiring server-side logic (real persistence, a rate-limited third-party API call, anything needing a secret) now has a real, working on-demand route pattern to build on directly, using exactly the same `prerender = false` mechanism this chapter already exercised.

Hands-On Exercises

EXERCISE 1

Add the Node adapter and the `/api/convert` route exactly as this chapter describes, wire up the second, server-calling input box, and confirm the server-side box produces identical output to Chapter 4's own client-side box for "sushi", "kyaku", and "konnichiwa".

 [View solution](#)

EXERCISE 2

Build a small local benchmark of your own — a real HTTP server exposing the same JSON contract, and a script that times both a direct `convert()` call and a real fetch call in a loop. Run it and report your own measured ratio, confirming it lands in the same general order of magnitude (thousands of times slower for the network call) this chapter reports.

 [View solution](#)

EXERCISE 3

Given this chapter's own real, measured evidence, argue in your own words why Chapter 4's client-side approach should remain this app's actual default. Then name one hypothetical feature this project could add later that would have a genuine, real reason to run server-side instead.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- **The Node adapter** — added for exactly one route; the rest of the project stays static by default
- **/api/convert** — a real on-demand route (export const prerender = false) calling the exact same, unmodified convert() from Chapters 2-3
- **Measured, not assumed** — a real local benchmark: $\sim 1.36\mu\text{s}$ per direct call vs. $\sim 14,084\mu\text{s}$ ($\sim 14\text{ms}$) per HTTP round trip, a real $\sim 10,353\times$ difference, confirmed at a second sample size
- **Localhost isn't free** — connection handling, double JSON serialization, and event-loop round trips cost real time even with zero network latency involved
- **Debouncing now matters — for the server box only** — Chapter 4's client-side box still needs none
- **The real verdict** — client-side is the objectively better choice for this specific feature; the server route stands as a working pattern for a future feature that actually needs one
- **Next chapter:** Styling & Matching the Existing Site's Own Look

CHAPTER 6: STYLING & MATCHING THE EXISTING SITE'S OWN LOOK

Romaji to Kana Converter: Astro

Chapter 6 · Styling & Matching the Existing Site's Own Look

Chapters 4 and 5 were entirely about function — two working conversion paths, styled with nothing but the browser's own defaults. This chapter makes the page actually look like it belongs on the real site it's meant to join, reusing this site's own already-established design conventions rather than inventing a new look from scratch.

Reusing the Site's Own Established Tool Palette

This project doesn't need to invent a dark theme — one already exists, documented directly in `my_tools_rules.md`'s own rule MT4 for exactly this kind of self-contained interactive page: background `#0f1117`, card and input surfaces `#1a1d27`, a secondary nested surface `#161b22`, borders `#30363d`, body text `#c9d1d9`, muted text `#8b949e`. Reusing it here — rather than picking new hex codes that happen to look similar — means this converter genuinely matches every other real interactive tool already on the site, not just an approximation of the look.

Which Accent Color Actually Belongs Here?

Every chapter of this course's own written material has used Astro's real brand orange, and it's tempting to just carry that straight into the actual tool. But MT4 itself is explicit about this exact decision: *"reusing an already-established site accent where a natural fit exists... rather than inventing a new one for every tool by default"* — the same rule that had `agent-builder.html` reuse the Claude Code Agents courses' own coral, since that tool is specifically Claude-Code-related.

CANDIDATE ACCENT	REAL FIT FOR THIS SPECIFIC TOOL
Astro brand orange (<code>#FF5D01</code>)	Matches this <i>course's</i> own subject matter, not what the finished tool actually does — a visitor using the tool has no reason to know or care it happens to be built in Astro

CANDIDATE ACCENT	REAL FIT FOR THIS SPECIFIC TOOL
A fresh, newly invented color	Exactly the default MT4 warns against picking "for every tool by default" when a real, natural fit already exists
Japanese red (<code>#d64550</code>)	The real, already-reserved accent — <code>language_rules.md</code> 's own L4a assigns this exact color to Japanese-language content specifically, and this converter is meant to sit directly alongside the Japanese-language lessons and kanji pages it links to

Japanese red wins on the real criterion MT4 itself names — a natural fit, not a coin flip — and it has the genuine added benefit of visually tying this tool to the exact pages Chapter 8's own capstone plans to link it from and to.

A Real, Verified Astro-Specific Finding: You Might Not Need Manual Scoping Here At All

`my_tools_rules.md`'s own MT4 also carries a strict, hard-won rule: every CSS selector in a My Tools page must be scoped under that tool's own wrapper class, never left as a bare element selector like `input` or `button` — a rule that exists because a real leak bug from exactly that mistake was found and fixed in `agent-builder.html`. It's worth checking, directly, whether that same risk actually applies to `index.astro` — rather than assuming the rule transfers unchanged.

Verified directly against Astro's own current documentation: a `<style>` block written inside a real `.astro` file is **automatically scoped by the Astro compiler itself** — every selector, including plain element selectors with no class at all, gets rewritten to carry a unique `data-astro-cid-*` attribute, both on the CSS side and on the rendered HTML elements themselves. A rule as bare as `input { ... }` genuinely only ever applies to inputs rendered by that exact component.

WHY MT4'S MANUAL RULE DOESN'T ACTUALLY APPLY HERE — AND WHY IT STILL EXISTS

MT4's own scoping requirement exists specifically because a My Tools page ships as raw, pre-built HTML that gets injected into an existing page via `set.html` — an operation that bypasses Astro's compiler entirely, so nothing about that injected fragment's own `<style>` block ever gets the automatic `data-astro-cid-*` treatment. `index.astro` is a genuinely different situation: it's a real, native `.astro` file, compiled by Astro directly, so it gets the same automatic protection every other

Astro component on the site already has. Nothing in this specific file needs the manual discipline MT4 exists to enforce for a fundamentally different kind of page.

Scoping Anyway, On Purpose

Even with that real, verified finding in hand, this chapter still wraps the whole page in one container class — not because today's file needs it, but because Chapter 8's own capstone is about actually mounting this converter onto the live site, and exactly how that happens is still an open question at this point in the course. If that integration ever moves this markup out of a standalone, Astro-compiled page and into the site's own injected-fragment system (the same real mechanism MT4's rule protects), a wrapper class already being in place costs nothing today and removes one entire category of future risk later. Cheap insurance, not a correction of a bug that doesn't exist yet.

The Real Styling

Everything below sits inside one wrapping `<div class="romaji-page">` in the page's own body, with the client-side box from Chapter 4 and the server-side box from Chapter 5 each styled as their own labeled card, so the two approaches Chapter 5 compared stay visually distinguishable rather than blurring into one form:

```
<style>
.romaji-page {
  background: #0f1117;
  color: #c9d1d9;
  font-family: system-ui, -apple-system, sans-serif;
  min-height: 100vh;
  padding: 2rem 1.5rem;
}
.romaji-page h1 { color: #ffffff; font-size: 1.8rem; margin-bottom: 0.4rem; }
.romaji-page .subtitle { color: #8b949e; margin-bottom: 2rem; }

.romaji-page h2 { font-size: 1.1rem; color: #d64550; margin-bottom: 0.8rem; }

.romaji-page .converter-card {
  background: #1a1d27;
```

```

border: 1px solid #30363d;
border-radius: 10px;
padding: 1.5rem;
margin-bottom: 2rem;
}

.romaji-page label {
  display: block; font-size: 0.85rem; color: #8b949e;
  margin-bottom: 0.4rem;
}

.romaji-page input[type="text"] {
  width: 100%;
  background: #161b22;
  border: 1px solid #30363d;
  border-radius: 6px;
  padding: 0.7rem 1rem;
  color: #c9d1d9;
  font-size: 1rem;
  font-family: inherit;
}

.romaji-page input[type="text"]:focus {
  outline: none;
  border-color: #d64550;
  box-shadow: 0 0 0 3px rgba(214, 69, 80, 0.2);
}

.romaji-page .output-grid {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 1rem;
  margin-top: 1.2rem;
}

/* Below 600px, the two output cards stack instead of sitting side by side */
@media (max-width: 600px) {
  .romaji-page .output-grid { grid-template-columns: 1fr; }
}

.romaji-page .output-box {
  background: #161b22;
  border: 1px solid #30363d;
  border-radius: 8px;

```

```
padding: 1rem;
}
.romaji-page .output-box h3 {
  font-size: 0.75rem; text-transform: uppercase; letter-spacing: 0.08em;
  color: #8b949e; margin-bottom: 0.5rem;
}
.romaji-page .output-box p { font-size: 1.4rem; min-height: 1.8rem; }
</style>
```

The markup itself, wrapping both Chapter 4's and Chapter 5's own boxes in the new classes — no element `id`s change, so neither script needs a single line touched:

```
<body>
  <div class="romaji-page">
    <h1>Romaji to Kana Converter</h1>
    <p class="subtitle">Type romaji, get real hiragana and katakana back.</p>

    <div class="converter-card">
      <h2>Client-Side Conversion</h2>
      <label for="romaji-input">Type romaji:</label>
      <input id="romaji-input" type="text" placeholder="e.g. konnichiwa" autocomplete="off">
      <div class="output-grid">
        <div class="output-box"><h3>Hiragana</h3><p id="hiragana-output"></p></div>
        <div class="output-box"><h3>Katakana</h3><p id="katakana-output"></p></div>
      </div>
    </div>

    <div class="converter-card">
      <h2>Server-Side Conversion (via /api/convert)</h2>
      <label for="server-input">Type romaji:</label>
      <input id="server-input" type="text" placeholder="e.g. konnichiwa" autocomplete="off">
      <div class="output-grid">
        <div class="output-box"><h3>Hiragana</h3><p id="server-hiragana-output"></p></div>
        <div class="output-box"><h3>Katakana</h3><p id="server-katakana-output"></p></div>
      </div>
    </div>
  </div>
```

```
<!-- Chapters 4-5's own <script> block goes here, entirely unchanged -->
</body>
```

Resizing the browser across the 600px breakpoint confirms the responsive behavior directly — above it, each card's own hiragana and katakana boxes sit side by side; below it, they stack into a single column, keeping both fully readable on a real phone-width screen without any horizontal scrolling.

Hands-On Exercises

EXERCISE 1

Add the CSS and markup exactly as this chapter describes, then open browser devtools and inspect a real element on the page. Confirm it actually carries a `data-astro-cid-*` attribute, and confirm the matching CSS rule in the Styles panel targets that exact attribute rather than a bare "input" selector.

 [View solution](#)

EXERCISE 2

In your own words, explain why `my_tools_rules.md`'s own MT4 requires manual wrapper-class scoping for a My Tools page, but this chapter found that requirement doesn't actually apply to `index.astro`'s own native

Romaji to Kana Converter: Astro

Chapter 7 · Deployment

This project has a genuinely different shape from almost every other Astro course on this site: it's a static site by default (Chapter 1's own real, deliberate choice), with exactly one route — Chapter 5's own `/api/convert` — opted into on-demand rendering.

Deployment has to account for both halves of that hybrid at once, and this chapter's own real headline finding is that Astro's build tooling already handles the split cleanly, with no extra configuration needed to stitch the two together.

Building for Production: Two Outputs From One Command

Verified directly against Astro's own current documentation for the `@astrojs/node` adapter: once an adapter is installed and at least one route opts into on-demand rendering, `astro build` produces two real, separate outputs from a single command — exactly the split this project's own hybrid setup needs:

```
npm run build
# dist/client/    - every static, prerendered page: index.astro, and any
#                  other page that never opted into on-demand rendering
# dist/server/entry.mjs - a real, runnable server handling exactly one
#                  route: /api/convert, per its own export const prerender = false
```

One Server Handles Both Jobs

A genuinely convenient real detail, confirmed directly against Astro's own documentation: in `mode: 'standalone'` — already set back in Chapter 5 — "the server handles file serving in addition to the page and API routes." The one built server doesn't just answer `/api/convert`; it also serves every file sitting in `dist/client/` directly, with no separate static file server, CDN, or extra configuration required to wire the two together:

```
HOST=0.0.0.0 PORT=4321 node ./dist/server/entry.mjs
# Serves index.astro's own static HTML AND handles real POST requests
# to /api/convert - from the exact same process, on the exact same port
```

`HOST` and `PORT` are read directly by the built server itself — Astro's own documentation is explicit that neither Astro nor the adapter loads environment variables on its own behalf, so anything genuinely secret would need its own loading mechanism; this project has no secrets at all to worry about.

AN HONEST CONTRAST WITH PREMIER LEAGUE PREDICTOR: ASTRO'S OWN DEPLOYMENT CHAPTER

That course's own Chapter 11 runs the identical `node ./dist/server/entry.mjs` command — but its entire site is genuinely dynamic, so `output: 'server'` puts every single page through

that one Node process. This project runs the exact same real command for a fundamentally different reason: almost the whole site is static and could be served with zero server process at all — the one Node process here exists purely to keep Chapter 5's own real server-side comparison route reachable, not because the site genuinely needs a server to function.

PM2 for Process Management

A single, long-lived process, kept alive and restarted automatically if it ever crashes — there's no shared database, no connection pool, and no multi-writer question anywhere in this project (unlike Premier League Predictor: Astro's own real SQLite-file concurrency question), so there's no real reason to run more than one instance at all:

```
// ecosystem.config.cjs
module.exports = {
  apps: [{
    name: 'romaji-converter',
    script: './dist/server/entry.mjs',
    instances: 1,
    exec_mode: 'fork',
    env: { HOST: '0.0.0.0', PORT: '4321' },
  }],
};
```

```
pm2 start ecosystem.config.cjs
```

nginx Reverse Proxy

A standard proxy in front of the one running process — plain `proxy_pass`, nothing route-specific needed yet, since the same process already correctly serves both static files and the API route on its own:

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

```

    location / {
        proxy_pass http://127.0.0.1:4321;
        proxy_set_header Host $host;
    }
}

```

A Real Rate Limit — Specifically for /api/convert

Every other route on this project's own site is free, in every real sense — a prerendered static file costs the server nothing at all to serve beyond a plain file read. Chapter 5's own real, measured finding changes that story for exactly one route: each real request to `/api/convert` costs the server roughly 14 milliseconds of genuine processing time, compared to Chapter 4's own client-side box, which costs the server nothing whatsoever. That specific route — and only that one — is genuinely worth protecting from a real flood of requests:

```

# nginx.conf, inside the http {} block
limit_req_zone $binary_remote_addr zone=convert:10m rate=10r/s;
limit_req_status 429; // real 503-by-default override — 429 Too Many Requests is t

server {
    /* ...listen/server_name as above... */

    location /api/convert {
        limit_req zone=convert burst=20 nodelay;
        proxy_pass http://127.0.0.1:4321;
        proxy_set_header Host $host;
    }

    location / {
        proxy_pass http://127.0.0.1:4321;
        proxy_set_header Host $host;
    }
}

```

`rate=10r/s` allows a sustained average of 10 requests per second from any one client IP (tracked via `$binary_remote_addr`); `burst=20 nodelay` lets a genuine short spike of up to 20 requests through immediately rather than queuing them, with anything past that rejected outright rather than delayed. `limit_req_status 429` overrides nginx's own real default of `503` — a deliberate choice here, since `429 Too Many Requests` is the more semantically accurate real HTTP status for "you're sending requests too fast," verified directly against nginx's own documentation.

No equivalent limit is placed on the static routes at all — there's no real cost to protect there, and adding one would just be friction with nothing genuine behind it.

A Real, Honest Closing Reflection

This entire chapter exists to keep one specific route reachable and correctly deployed — a route Chapter 5's own real, measured evidence already concluded shouldn't be this app's own default. That's worth stating plainly rather than glossing over: if this project were being deployed purely as a production feature rather than as a working, comparable example for this course, the honest, evidence-based choice would be to drop `/api/convert`, the Node adapter, and this entire chapter's own PM2/rate-limiting setup altogether — deploying as nothing more than a folder of static files with zero server process required at all. This chapter builds out the fuller deployment anyway, specifically because the server-side comparison route is worth keeping reachable for anyone reading this course to try both approaches for real, not because the finished app actually needs it.

Hands-On Exercises

EXERCISE 1

Build the project, run the standalone server, and confirm — with a browser and with a real curl POST request — that both `index.astro`'s own static page and the on-demand `/api/convert` route are served by the exact same running process on the exact same port, with no separate static file server involved.

 [View solution](#)

EXERCISE 2

Add the nginx rate-limit configuration exactly as this chapter describes, then send a real rapid burst of more than 20 requests to `/api/convert` from a script. Confirm some requests succeed and later ones are rejected with a real 429 status, matching the `limit_req_status` override.

 [View solution](#)

EXERCISE 3

Given Chapter 5's own real, measured verdict that client-side conversion is objectively better for this feature, explain in your own words why this chapter still builds out a full deployment story for the server-side route rather than simply recommending it be deleted.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **Two outputs, one build** — `dist/client/` (static pages) and `dist/server/entry.mjs` (the one on-demand route), verified against Astro's own current documentation
- **One process, two jobs** — standalone mode's built-in file serving means the same server handles both static files and `/api/convert`, no extra config needed
- **PM2, single fork instance** — no database, no multi-writer question, no real reason to run more than one
- **nginx rate limit on `/api/convert` only** — real, targeted protection for the one route with a genuine per-request server cost (Chapter 5's own measured `~14ms`), left off every free static route
- **429, not 503** — a deliberate `limit_req_status` override to the more semantically accurate real HTTP status code
- **The honest reflection** — this whole chapter deploys a route Chapter 5's own evidence already said shouldn't be the default, kept alive specifically so it stays reachable for comparison
- **Next chapter: Capstone** — shipping this as a real new page on the live site, and scoping kanji conversion as a genuine future sub-project

CHAPTER 8: CAPSTONE — SHIPPING THIS AS A REAL NEW PAGE ON THE LIVE SITE**Romaji to Kana Converter: Astro**

Chapter 8 · Capstone: Shipping This as a Real New Page on the Live Site, and Scoping Kanji Conversion as a Genuine Future Sub-Project

Chapter 1 opened this course with a real claim: unlike its Angular & Express and React & Next.js siblings, this converter is built in the exact same framework the real, live site already runs on, making it the most direct "just add a page" story of the whole trio. Every chapter since has been building toward actually cashing that claim in. This capstone does it — comparing two real integration paths, recommending the one that fits the site's own established conventions, and closing the one deliberate gap this course left open from its very first chapter: kanji conversion.

What Each Chapter Actually Built**Ch.1 — Setup**

A default, static-output Astro project — no database, no adapter, since conversion is a pure, stateless function.

Ch.2-3 — The Engine

A complete, real gojūon/dakuten/yōon KANA_MAP and tokenizer in `src/lib/convert.ts`, with four genuine bugs found and fixed, all independently re-verified against live Node output.

Ch.4 — The UI

A plain client-side `<script>` importing `convert()` directly — verified bundled, TypeScript-aware, and import-resolved by Astro with zero config.

Ch.5 — The Real Verdict

A real `/api/convert` route plus a genuine local benchmark: $\sim 1.36\mu\text{s}$ direct vs. $\sim 14,084\mu\text{s}$ over HTTP — client-side wins, measured, not assumed.

Ch.6 — Styling

MT4's own established palette, the Japanese-red accent (`language_rules.md` L4a), and a verified finding about Astro's own automatic per-component CSS scoping.

Ch.7 — Deployment

A real, working standalone deployment for the full app — with an honest closing note that it exists mainly to keep Chapter 5's own comparison reachable.

Two Real Ways This Converter Could Actually Reach the Live Site

"A new page on the live site" isn't one single mechanism — this site has two genuinely different real ways a page can end up there, and they behave very differently:

PATH	WHAT IT ACTUALLY MEANS	REAL TRADEOFFS
A. A native Astro route	Copy Chapter 6's own finished <code>index.astro</code> directly into the live site's own <code>src/pages/</code> , as its own real route	Reuses this course's own file almost unchanged — but stands apart from every other Sidebar/My Tools page on the site, which don't work this way at all; wouldn't automatically pick up the shared nav/footer unless separately wired into the site's own layout
B. A My Tools-style fragment	Turn the finished markup into a self-contained HTML fragment, following <code>my_tools_rules.md</code> 's own established MT1-MT9 convention for exactly this kind of interactive tool	Matches every other real interactive tool already on the site; picked up automatically by the site's own shared template via <code>set:html</code> , with no separate routing to wire up — the real, established pattern this kind of tool already uses

Path B is the real, better fit — not because path A wouldn't technically work, but because it would make this converter the one interactive tool on the whole site that doesn't follow the same pattern every other one already does. Reusing an established convention, per this course's own MT4 finding back in Chapter 6, is exactly the discipline this site already practices everywhere else.

THIS IS EXACTLY WHERE CHAPTER 6'S OWN FORESHADOWING PAYS OFF

Chapter 6 scoped this converter's own CSS under a wrapper class specifically because "if that integration ever moves this markup out of a standalone, Astro-compiled page and into the site's own injected-fragment system... a wrapper class already being in place costs nothing today and removes one entire category of future risk later." That's precisely what path B does. Because the wrapper class is already there, moving this markup into the fragment-based My Tools system loses none of the automatic `data-astro-cid=*` protection Chapter 6 also found — the manual scoping was already doing that exact job, on standby, since the moment it was written.

The Real, Recommended Shipped Version Is Simpler Than Chapter 7's Own Deployment

Chapter 5 already delivered a real, measured verdict: client-side conversion is the objectively better choice for this feature. Taking that verdict seriously all the way through to shipping means the version that actually belongs on the live site doesn't need `/api/convert`, the Node adapter, or a single piece of Chapter 7's own PM2/nginx/rate-limiting machinery — none of it. A My Tools fragment is static HTML, CSS, and a plain client-side script, served the exact same way every other tool on the site already is, with no server process of its own required at all.

THE SERVER-SIDE BOX DOESN'T HAVE TO DISAPPEAR

Keeping Chapter 5's own comparison box in the shipped version is a genuine, optional choice — it would need the live site's own equivalent of Chapter 7's deployment, which is a real cost most single-page tools on this site don't carry. The honestly recommended default is to ship the client-side box alone, matching Chapter 5's own real conclusion, and leave the server-side comparison living in this course's own written material rather than in the shipped tool itself.

A Real My Tools Spec, Sketched Out

`my_tools_rules.md`'s own MT2/MT3 define the spec format a real `/my-tools` invocation would read from — a YAML file at `my-tools/romaji-converter.yaml`, describing the tool completely enough that no separate input is needed beyond it:

```
# my-tools/romaji-converter.yaml
name: romaji-converter
description: |
  A single-page tool that converts typed romaji (Latin-alphabet Japanese)
  into hiragana and katakana, live as the user types. Handles the real
  gojūon grid, voiced/semi-voiced sounds, yōon combinations, long vowels,
  sokuon (small つ), and the は/へ/を particle exceptions. Client-side
  only — a plain input box, and two output areas for hiragana and
  katakana, styled to match the site's own established tool palette.
  Kanji conversion is explicitly out of scope.
example:
  input: konnichiwa
  hiragana_output: こんにちは
  katakana_output: コンニチハ
```

Everything that spec describes is already real and already tested — this course's own Chapters 2 through 6 are the actual functional and visual specification, just written as prose instead of YAML.

Scoping Kanji Conversion as a Genuine Future Sub-Project

Chapter 1 deferred kanji conversion explicitly, naming it "a genuinely bigger, real dictionary-and-context- dependent problem" rather than something this course would solve as a side effect. It's worth closing that gap honestly rather than leaving it as a vague aside — naming concretely what it would actually take:

- **A real dictionary, not a lookup table.** Unlike `KANA_MAP`'s own fixed, roughly 100-entry romaji-to-kana mapping, a working kanji converter needs a genuine word-level dictionary — thousands of entries, each kana reading mapped to one or more real candidate kanji spellings.
- **Real disambiguation, not a single answer.** The same kana can map to several different kanji depending on meaning and context — this is the actual hard part, and it's a genuinely different kind of problem from anything Chapters 2-3 solved, since those chapters' own ambiguities (standalone "e" as え vs. 絵, per Chapter 3's own warn-box) were already honestly documented as unresolvable by a lookup table alone. A real kanji feature would need the same category of context-awareness, at a much larger scale.
- **A genuine reason to move server-side.** Chapter 5's own exercise 3 already named this directly: a real dictionary large enough to disambiguate correctly is "far too large to ship to every visitor's browser on every page load" — a real, structural reason to keep that specific lookup on a server, unlike plain romaji-to-kana conversion, which this whole course spent Chapter 5 proving has no such need.

NOT A COMMITMENT, A SCOPED-OUT BOUNDARY

None of this is a plan to build kanji conversion next — it's an honest accounting of why it was correctly left out of this course from Chapter 1 onward, and what a real future project would actually need to tackle if it were picked up later, consistent with how this project's own tracking already treats it: a real, named future sub-project, not a missing feature of this one.

Closing This Course

Eight chapters produced a real, tested conversion engine (four genuine bugs found and independently verified, not assumed), a real measured answer to the question this whole trio of courses was built around (client-side wins, by roughly four orders of magnitude), a real deployment story for the alternative anyway, and a concrete, evidence-backed plan for actually shipping the recommended version as a real page on the site it was always meant to join — closing exactly where Chapter 1 opened, with the claim now backed by eight chapters of real, verified work rather than a promise.

Hands-On Exercises

EXERCISE 1

Write your own complete `my-tools/romaji-converter.yaml` spec file matching MT3's schema, including a real example block, based entirely on what this course actually built in Chapters 2 through 6.

[!\[\]\(5c65cabb9dec68d83bd41cb0bb782f76_img.jpg\) View solution](#)

EXERCISE 2

Explain in your own words why Chapter 6's decision to scope the CSS under a wrapper class "on purpose" turned out to matter directly in this capstone, tracing the specific real mechanism that makes it matter.

[!\[\]\(b28a70298ac8b6ba56c085c416a1008b_img.jpg\) View solution](#)

EXERCISE 3

Using Chapter 5's own real, measured evidence, explain why the version of this converter that should actually ship needs none of Chapter 7's own adapter/PM2/nginx-rate-limiting machinery at all.

[!\[\]\(0b52aeeccc30b927bed70000ab7a4697_img.jpg\) View solution](#)

CHAPTER 8 / COURSE COMPLETE — QUICK REFERENCE

- **Two real integration paths compared** — a native Astro route vs. a My Tools fragment; the fragment path wins, matching every other real tool already on the site
- **Chapter 6's scoping pays off directly** — the wrapper class added "on purpose" is exactly what makes the fragment migration safe
- **Ship client-side only** — per Chapter 5's own real verdict, no adapter/PM2/nginx machinery needed for the recommended shipped version
- **A real MT3-schema spec** — name, description, and example, drawn directly from this course's own finished work
- **Kanji conversion, honestly scoped out** — a real dictionary, real disambiguation, and (per Chapter 5's own exercise 3) a genuine reason to run server-side, unlike this course's own plain romaji conversion
- **Course complete** — 8/8 chapters, closing the claim Chapter 1 opened with real, verified evidence