

A HANDS-ON TUTORIAL

Translating FHIR Data

Store localized labels on Observations at ingestion, and switch language in the browser
Translations are written during ingestion and can be refreshed via update routes; the browser only switches which one is shown.

Panagiotis Liakakos

June 2026

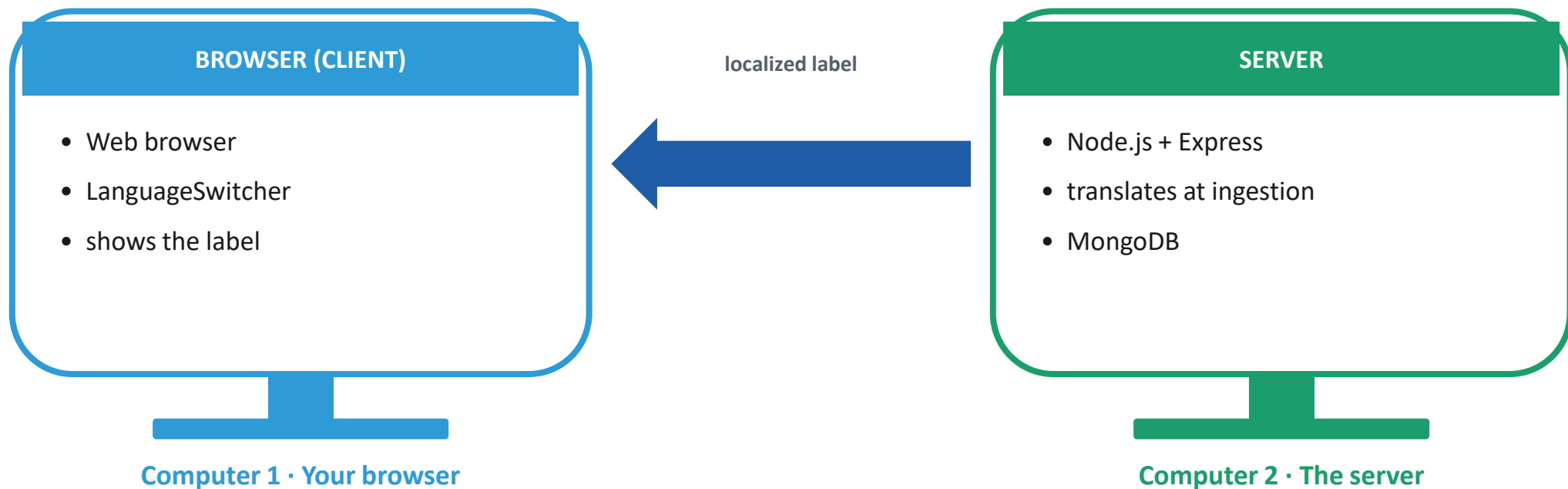
What you'll learn

By the end, you will be able to:

- Load official LOINC translations from files on the server.
- Build cached per-language maps from LOINC code to a localized label.
- Store a translation in per-language fields on each Observation, at ingestion.
- Refresh them via update routes, and switch language in the browser.

Goal: label Observations by their LOINC code, so the browser can just switch language.

How the data flows: server → browser



The LOINC translation files, from loinc.org, live on the server. As Observations are ingested, the server stores localized labels on them (and update routes can refresh them). The browser then switches language and shows the matching label.

What you need to translate FHIR data



You only need three tools

- The LOINC Linguistic Variant CSVs + abbreviations.csv in backend/data/csv/ (from loinc.org).
- csv-parser + Mongoose on the server — read the files and store labels on Observations.
- A LanguageContext on the client — to switch the displayed language.

Prerequisites: Node 18+, MongoDB, and the LOINC Linguistic Variant files downloaded from loinc.org (Regenstrief) into backend/data/csv/.
Translations are set at ingestion (observationService.js) and refreshed via translationService.js update routes.

Step 1 — Build the translation maps (server)

IN PLAIN ENGLISH

- Load abbreviations.csv into a PROPERTY → full-label map.

backend/services/translationService.js

```
// backend — build one language's loinc -> label map (csv-parser)
async function buildTranslationMap(cfg) {
  const propMap = await loadPropertyMap(); // abbrev -> full
  const map = {};

  await new Promise((res) =>
    fs.createReadStream(cfg.csv).pipe(csv()) // csv-parser
      .on("data", (row) => {
        const code = row[cfg.codeCol]?.trim(); // LOINC_NUM
        const comp = row[cfg.textCol]?.trim(); // COMPONENT
        if (!code || !comp) return;
        const rawProp = (row[cfg.propertyCol] || "").trim();
        const full = propMap[canon(rawProp)]; // MCNC -> ...
        const meth = row[cfg.methodCol]?.trim() || "";
        map[code] = (comp + (full ? ` ${full}` : "")
          + (meth ? ` ${meth}` : "")).trim();
      })
      .on("end", res));

  return map; // cached per language by the caller
}
```

Step 1 — Build the translation maps (server)

IN PLAIN ENGLISH

- Load abbreviations.csv into a PROPERTY → full-label map.
- Read the language's LOINC Linguistic Variant CSV with csv-parser.

backend/services/translationService.js

```
// backend — build one language's loinc -> label map (csv-parser)
async function buildTranslationMap(cfg) {
  const propMap = await loadPropertyMap(); // abbrev -> full
  const map = {};

  await new Promise((res) =>
    fs.createReadStream(cfg.csv).pipe(csv()) // csv-parser
      .on("data", (row) => {
        const code = row[cfg.codeCol]?.trim(); // LOINC_NUM
        const comp = row[cfg.textCol]?.trim(); // COMPONENT
        if (!code || !comp) return;
        const rawProp = (row[cfg.propertyCol] || "").trim();
        const full = propMap[canon(rawProp)]; // MCNC -> ...
        const meth = row[cfg.methodCol]?.trim() || "";
        map[code] = (comp + (full ? ` ${full}` : "")
          + (meth ? ` ${meth}` : "")).trim();
      })
      .on("end", res));

  return map; // cached per language by the caller
}
```

Step 1 — Build the translation maps (server)

IN PLAIN ENGLISH

- Load abbreviations.csv into a PROPERTY → full-label map.
- Read the language's LOINC Linguistic Variant CSV with csv-parser.
- **Build a label (component + expanded property + method), keyed by LOINC code.**

backend/services/translationService.js

```
// backend — build one language's loinc -> label map (csv-parser)
async function buildTranslationMap(cfg) {
  const propMap = await loadPropertyMap(); // abbrev -> full
  const map = {};

  await new Promise((res) =>
    fs.createReadStream(cfg.csv).pipe(csv()) // csv-parser
    .on("data", (row) => {
      const code = row[cfg.codeCol]?.trim(); // LOINC_NUM
      const comp = row[cfg.textCol]?.trim(); // COMPONENT
      if (!code || !comp) return;
      const rawProp = (row[cfg.propertyCol] || "").trim();
      const full = propMap[canon(rawProp)]; // MCNC -> ...
      const meth = row[cfg.methodCol]?.trim() || "";
      map[code] = (comp + (full ? ` ${full}` : "")
        + (meth ? ` ${meth}` : "")).trim();
    })
    .on("end", res));

  return map; // cached per language by the caller
}
```

Step 1 — Build the translation maps (server)

IN PLAIN ENGLISH

- Load abbreviations.csv into a PROPERTY → full-label map.
- Read the language's LOINC Linguistic Variant CSV with csv-parser.
- Build a label (component + expanded property + method), keyed by LOINC code.
- **Cache the finished map — one per language.**

backend/services/translationService.js

```
// backend — build one language's loinc -> label map (csv-parser)
async function buildTranslationMap(cfg) {
  const propMap = await loadPropertyMap(); // abbrev -> full
  const map = {};

  await new Promise((res) =>
    fs.createReadStream(cfg.csv).pipe(csv()) // csv-parser
    .on("data", (row) => {
      const code = row[cfg.codeCol]?.trim(); // LOINC_NUM
      const comp = row[cfg.textCol]?.trim(); // COMPONENT
      if (!code || !comp) return;
      const rawProp = (row[cfg.propertyCol] || "").trim();
      const full = propMap[canon(rawProp)]; // MCNC -> ...
      const meth = row[cfg.methodCol]?.trim() || "";
      map[code] = (comp + (full ? ` ${full}` : "")
        + (meth ? ` ${meth}` : "")).trim();
    })
    .on("end", res));

  return map; // cached per language by the caller
}
```

The Observation model — where translations live

IN PLAIN ENGLISH

- Each Observation stores its LOINC code and English testName.
- Plus one field per language: greekTranslation, italianTranslation, ...
- These fields are filled at ingestion (and by the update routes).
- The client reads the field for the selected language.

backend/models/Observation.js

```
// backend/models/Observation.js (Mongoose schema)
const ObservationSchema = new Schema({
  code:      CodeableConceptSchema, // includes the LOINC code
  testName:  String,                // English name
  value:     Number,
  unit:      String,
  date:      Date,

  // one field per language (filled at ingestion / refresh)
  greekTranslation: String,
  italianTranslation: String,
  germanTranslation: String,
  spanishTranslation: String,
  frenchTranslation: String,
  dutchTranslation: String,

  rawFhir: Schema.Types.Mixed,
});
```

Step 2 — Translate at ingestion (server)

IN PLAIN ENGLISH

- A helper looks each LOINC code up in every language map.

backend/services/observationService.js

```
// backend – translate as each Observation is ingested
async function getTranslationsForCode(code) {
  const out = {};
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const map = await loadTranslationMap(lang);
    out[lang] = map[code] || "";
  }
  return out;
}

async function processObservation(obs) {
  // ... resolve patient, value, date, category ...
  const loinc = obs.code?.coding?.[0]?.code;
  const translations = await getTranslationsForCode(loinc);

  const doc = { observationId, patientId, testName, value };
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const { docField } = LANGUAGE_CONFIG[lang];
    doc[docField] = translations[lang] || "";
  }
  await Observation.findOneAndUpdate(
    { observationId, testName }, { $set: doc }, { upsert: true });
}

// component Observations: each component upserted + translated
```

Step 2 — Translate at ingestion (server)

IN PLAIN ENGLISH

- A helper looks each LOINC code up in every language map.
- **As an Observation is ingested, processObservation reads its LOINC code.**

backend/services/observationService.js

```
// backend – translate as each Observation is ingested
async function getTranslationsForCode(code) {
  const out = {};
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const map = await loadTranslationMap(lang);
    out[lang] = map[code] || "";
  }
  return out;
}

async function processObservation(obs) {
  // ... resolve patient, value, date, category ...
  const loinc = obs.code?.coding?.[0]?.code;
  const translations = await getTranslationsForCode(loinc);

  const doc = { observationId, patientId, testName, value };
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const { docField } = LANGUAGE_CONFIG[lang];
    doc[docField] = translations[lang] || "";
  }
  await Observation.findOneAndUpdate(
    { observationId, testName }, { $set: doc }, { upsert: true });
}
// component Observations: each component upserted + translated
```

Step 2 — Translate at ingestion (server)

IN PLAIN ENGLISH

- A helper looks each LOINC code up in every language map.
- As an Observation is ingested, processObservation reads its LOINC code.
- **It fills each language's field and upserts the Observation with its translations.**

backend/services/observationService.js

```
// backend – translate as each Observation is ingested
async function getTranslationsForCode(code) {
  const out = {};
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const map = await loadTranslationMap(lang);
    out[lang] = map[code] || "";
  }
  return out;
}

async function processObservation(obs) {
  // ... resolve patient, value, date, category ...
  const loinc = obs.code?.coding?.[0]?.code;
  const translations = await getTranslationsForCode(loinc);

  const doc = { observationId, patientId, testName, value };
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const { docField } = LANGUAGE_CONFIG[lang];
    doc[docField] = translations[lang] || "";
  }
  await Observation.findOneAndUpdate(
    { observationId, testName }, { $set: doc }, { upsert: true });
}

// component Observations: each component upserted + translated
```

Step 2 — Translate at ingestion (server)

IN PLAIN ENGLISH

- A helper looks each LOINC code up in every language map.
- As an Observation is ingested, processObservation reads its LOINC code.
- It fills each language's field and upserts the Observation with its translations.
- **For component Observations, each component is stored and translated separately.**

backend/services/observationService.js

```
// backend – translate as each Observation is ingested
async function getTranslationsForCode(code) {
  const out = {};
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const map = await loadTranslationMap(lang);
    out[lang] = map[code] || "";
  }
  return out;
}

async function processObservation(obs) {
  // ... resolve patient, value, date, category ...
  const loinc = obs.code?.coding?.[0]?.code;
  const translations = await getTranslationsForCode(loinc);

  const doc = { observationId, patientId, testName, value };
  for (const lang of Object.keys(LANGUAGE_CONFIG)) {
    const { docField } = LANGUAGE_CONFIG[lang];
    doc[docField] = translations[lang] || "";
  }
  await Observation.findOneAndUpdate(
    { observationId, testName }, { $set: doc }, { upsert: true });
}

// component Observations: each component upserted + translated
```

Refresh & backfill — the update routes

IN PLAIN ENGLISH

- Three POST routes re-run translation after ingestion.
- update-all re-translates every Observation; update-many takes a list of ids.
- update-one/:observationId re-translates a single Observation.
- Each accepts an optional languages list in the body.

backend/routes/translationRoutes.js + translationController.js

```
// backend/routes/translationRoutes.js
router.post("/update-all", updateTranslationsForAllObservations);
router.post("/update-many", updateTranslationsByIds);
router.post("/update-one/:observationId",
            updateTranslationForOneObservation);
```

```
// body (optional): { languages: ["el","it"] }
// re-runs the same tagging as ingestion, then reports
// how many Observations were updated
```

```
// POST /api/translations/update-all
// POST /api/translations/update-many { observationIds }
// POST /api/translations/update-one/<id>
```

On the client — how it's displayed

IN PLAIN ENGLISH

- LanguageContext holds the selected language (default English).
- The LanguageSwitcher dropdown changes it.
- ObservationsChart maps the language to a translation field.
- It shows that field, falling back to the English testName.

```
frontend · LanguageSwitcher / ObservationsChart
```

```
// frontend - LanguageSwitcher + ObservationsChart
const { language } = useContext(LanguageContext); // "en" default

// the switcher changes the language
<select value={language}
  onChange={(e) => toggleLanguage(e.target.value)}>
  <option value="en">EN</option>
  <option value="gr">GR</option>  {/* it, de, es, fr, nl */}
</select>

// ObservationsChart maps language -> field, then renders it
const FIELD_MAP = { gr: "greekTranslation",
                    it: "italianTranslation", /* ... */ };
const field = FIELD_MAP[language];
const title = field && obs[field] ? obs[field] : obs.testName;
```

What you learned

The whole pipeline, in six moves

- LOINC translation files (from loinc.org) live in `backend/data/csv/`.
- `translationService` builds cached per-language maps with `csv-parser`.
- Observations store a field per language (`greekTranslation`, ...).
- At ingestion, `observationService` fills those fields by LOINC code.
- Update routes re-run the tagging to refresh or backfill.
- The `LanguageSwitcher` selects the language, and `ObservationsChart` shows the matching field.

Translation is mostly a backend concern: label Observations by LOINC code, then just switch language in the browser.