

A HANDS-ON TUTORIAL

Reading FHIR Data

Ingest FHIR resources and store them in a MongoDB database

Panagiotis Liakakos

June 2026

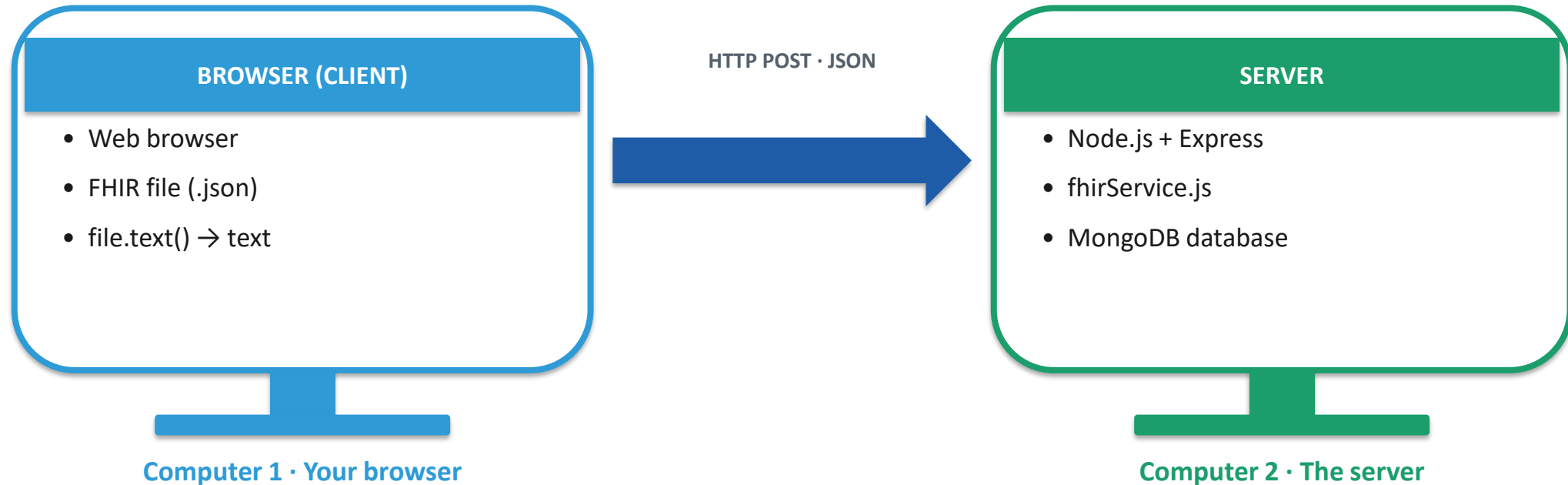
What you'll learn

By the end, you will be able to:

- Read a FHIR file from disk into your app — entirely in the browser.
- Send the raw JSON to a Node / Express backend and parse it there.
- Walk a FHIR Bundle in the right order and route each supported resource to a handler.
- Store each supported resource as one or more MongoDB documents — keeping the original intact.

Goal: ingest FHIR data and store it in a MongoDB database.

How the data travels: browser → server



FHIR data starts as a file on the user's computer, where the browser reads it, then travels over HTTP to the server, which parses it and stores each supported resource in MongoDB.

What you need to read FHIR data

You only need three tools

- Browser File API — `file.text()` reads an upload into a string.
- Express + `express.json({ limit })` — receives and parses the JSON.
- Mongoose — one model per supported resource type stores it.
- No special FHIR library needed just to ingest — a Bundle is plain JSON.

Prerequisites: Node 18+ and a running MongoDB (mongod) — then the three tools above work.

The core ingest is ~100 lines in `fhirService.js` (~230 with the resource-dispatch table), plus a small model + service per resource type.

What you need to read FHIR data

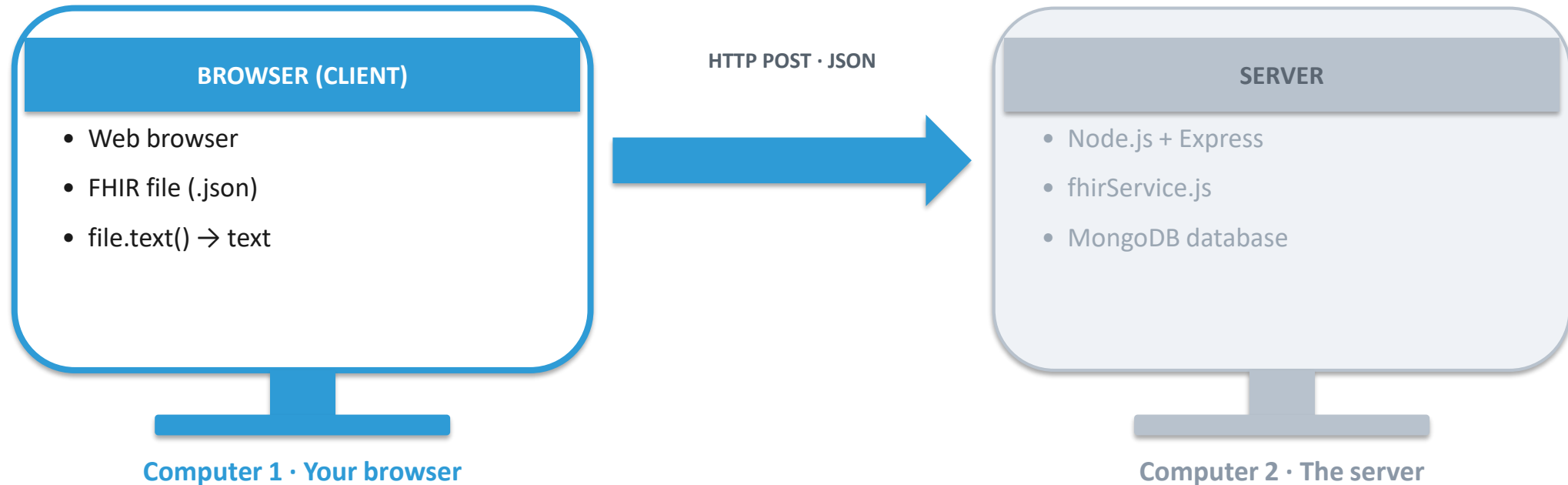
Five things not to forget

- Save in dependency order — a Patient before anything that references it.
- References arrive as urn:uuid:... or Type/id — strip both.
- Use upsert — it updates matching records and helps prevent duplicates.
- Keep the raw resource (rawFhir) so you can re-export later.
- Raise the body limit — real bundles are megabytes, not kilobytes.

Prerequisites: Node 18+ and a running MongoDB (mongod) — then the three tools above work.

The core ingest is ~100 lines in fhirService.js (~230 with the resource-dispatch table), plus a small model + service per resource type.

Next: the browser side (Computer 1)



The code in the next slides runs on Computer 1 — inside the user's web browser. It reads the chosen file and turns it into text, then sends it.

Step 1 — From the disk into memory (browser)

IN PLAIN ENGLISH

- The file is never uploaded as a “file” — we read its text in the browser first, then send that text.

frontend · FileUpload.jsx + App.jsx

```
// A plain file input gives us browser File objects
<input type="file" multiple accept=".json"
  onChange={handleFileUpload} />
```

```
async function handleFileUpload(event) {
  const files = Array.from(event.target.files);
  for (const file of files) {
```

```
    // Read the WHOLE file into a string:
    // disk -> memory, as a Promise<string>.
    const jsonText = await file.text();
```

```
    // ...now send jsonText to the server (Step 2)
  }
}
```

Step 1 — From the disk into memory (browser)

IN PLAIN ENGLISH

- The file is never uploaded as a “file” — we read its text in the browser first, then send that text.
- `<input type="file" accept=".json" multiple>` hands us File objects.

frontend · FileUpload.jsx + App.jsx

```
// A plain file input gives us browser File objects
<input type="file" multiple accept=".json"
      onChange={handleFileUpload} />

async function handleFileUpload(event) {
  const files = Array.from(event.target.files);
  for (const file of files) {

    // Read the WHOLE file into a string:
    // disk -> memory, as a Promise<string>.
    const jsonText = await file.text();

    // ...now send jsonText to the server (Step 2)
  }
}
```

Step 1 — From the disk into memory (browser)

IN PLAIN ENGLISH

- The file is never uploaded as a “file” — we read its text in the browser first, then send that text.
- `<input type="file" accept=".json" multiple>` hands us File objects.
- **`file.text()` (the Blob API) returns the whole file as a string — no FileReader, no callbacks.**

frontend · FileUpload.jsx + App.jsx

```
// A plain file input gives us browser File objects
<input type="file" multiple accept=".json"
  onChange={handleFileUpload} />

async function handleFileUpload(event) {
  const files = Array.from(event.target.files);
  for (const file of files) {

    // Read the WHOLE file into a string:
    // disk -> memory, as a Promise<string>.
    const jsonText = await file.text();

    // ...now send jsonText to the server (Step 2)
  }
}
```

Step 1 — From the disk into memory (browser)

IN PLAIN ENGLISH

- The file is never uploaded as a “file” — we read its text in the browser first, then send that text.
- `<input type="file" accept=".json" multiple>` hands us File objects.
- `file.text()` (the Blob API) returns the whole file as a string — no FileReader, no callbacks.
- **One file or many: we just loop over them.**

frontend · FileUpload.jsx + App.jsx

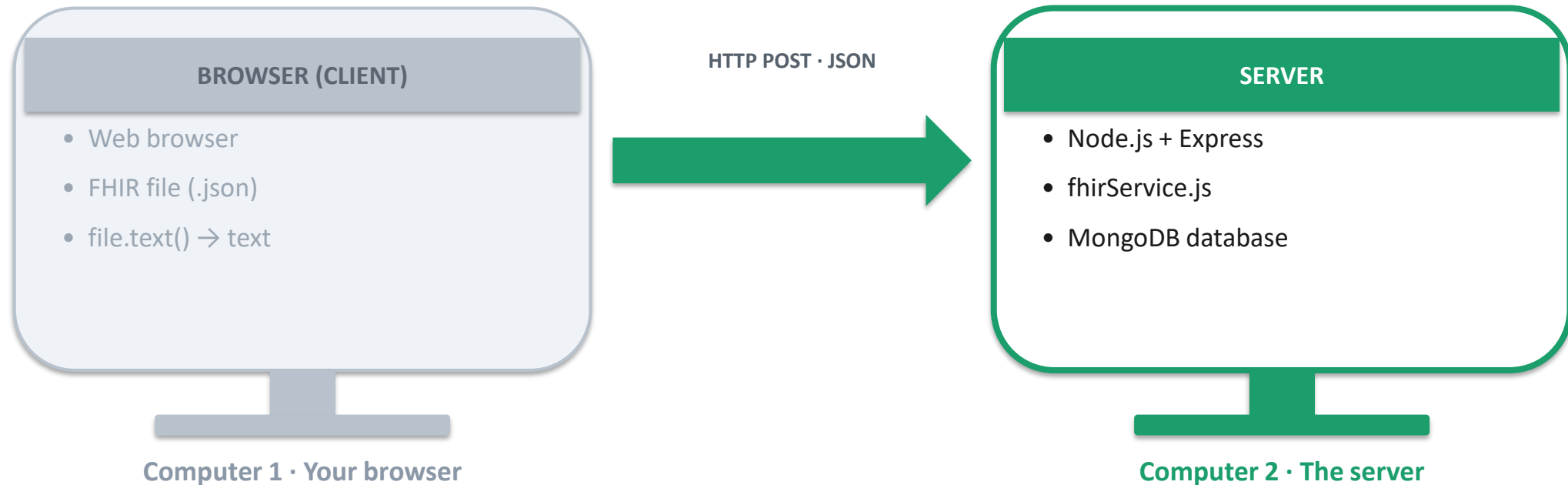
```
// A plain file input gives us browser File objects
<input type="file" multiple accept=".json"
  onChange={handleFileUpload} />

async function handleFileUpload(event) {
  const files = Array.from(event.target.files);
  for (const file of files) {

    // Read the WHOLE file into a string:
    // disk -> memory, as a Promise<string>.
    const jsonText = await file.text();

    // ...now send jsonText to the server (Step 2)
  }
}
```

Now: the server side (Computer 2)



The focus now moves to Computer 2. The next slides show the request arriving at the Node and Express server, where it is parsed and processed.

Step 2 — Over the wire, parsed on the server

IN PLAIN ENGLISH

- We POST the raw JSON string; the original filename rides along in an x-filename header.

```
frontend App.jsx -> backend server.js / fhirController.js

// FRONTEND – send the text we just read
await fetch(`${API}/fhir/upload`, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "x-filename": file.name,    // keep original name
  },
  body: jsonText,              // the raw JSON string
  credentials: "include",      // send session cookie
});

// BACKEND – server.js: ONE line parses every upload
app.use(express.json({ limit: "50mb" }));
// -> req.body is now a real JS object

// controllers/fhirController.js
const filename = req.headers["x-filename"];
await fhirService.uploadFhirData(req.body, filename);
```

Step 2 — Over the wire, parsed on the server

IN PLAIN ENGLISH

- We POST the raw JSON string; the original filename rides along in an x-filename header.
- **credentials:"include" sends the session cookie — needed by the protected read endpoints; the upload route itself is currently open.**

```
frontend App.jsx -> backend server.js / fhirController.js

// FRONTEND — send the text we just read
await fetch(`${API}/fhir/upload`, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "x-filename": file.name,    // keep original name
  },
  body: jsonText,              // the raw JSON string
  credentials: "include",      // send session cookie
});

// BACKEND — server.js: ONE line parses every upload
app.use(express.json({ limit: "50mb" }));
// -> req.body is now a real JS object

// controllers/fhirController.js
const filename = req.headers["x-filename"];
await fhirService.uploadFhirData(req.body, filename);
```

Step 2 — Over the wire, parsed on the server

IN PLAIN ENGLISH

- We POST the raw JSON string; the original filename rides along in an x-filename header.
- credentials:"include" sends the session cookie — needed by the protected read endpoints; the upload route itself is currently open.
- **On the server, `express.json()` turns the body back into a real JS object (`req.body`) — that is the parse step.**

```
frontend App.jsx -> backend server.js / fhirController.js

// FRONTEND – send the text we just read
await fetch(`${API}/fhir/upload`, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "x-filename": file.name,    // keep original name
  },
  body: jsonText,              // the raw JSON string
  credentials: "include",      // send session cookie
});

// BACKEND – server.js: ONE line parses every upload
app.use(express.json({ limit: "50mb" }));
// -> req.body is now a real JS object

// controllers/fhirController.js
const filename = req.headers["x-filename"];
await fhirService.uploadFhirData(req.body, filename);
```

Step 2 — Over the wire, parsed on the server

IN PLAIN ENGLISH

- We POST the raw JSON string; the original filename rides along in an x-filename header.
- `credentials:"include"` sends the session cookie — needed by the protected read endpoints; the upload route itself is currently open.
- On the server, `express.json()` turns the body back into a real JS object (`req.body`) — that is the parse step.
- **Bundles are big, so we raise the limit to 50 MB.**

```
frontend App.jsx -> backend server.js / fhirController.js

// FRONTEND — send the text we just read
await fetch(`${API}/fhir/upload`, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "x-filename": file.name,    // keep original name
  },
  body: jsonText,              // the raw JSON string
  credentials: "include",      // send session cookie
});

// BACKEND — server.js: ONE line parses every upload
app.use(express.json({ limit: "50mb" }));
// -> req.body is now a real JS object

// controllers/fhirController.js
const filename = req.headers["x-filename"];
await fhirService.uploadFhirData(req.body, filename);
```

Step 3 — A Bundle is just a list (order matters)

IN PLAIN ENGLISH

- A FHIR Bundle wraps everything in `entry[]`; each entry has a `.resource`.

backend/services/fhirService.js

```
// Accept a whole Bundle OR a single bare resource
const entries = Array.isArray(fhirData.entry)
  ? fhirData.entry           // Bundle -> entry list
  : [{ resource: fhirData }]; // single resource

// Lower number = stored earlier (dependencies first)
const processingOrder = {
  Patient: 1, Organization: 2, Practitioner: 2,
  Observation: 5, Condition: 6, Encounter: 7,
  DiagnosticReport: 8, // needs its Observations first
  Composition: 10,    // references everything -> last
};

// Unknown types sink to the bottom (999)
const sorted = entries.slice().sort((a, b) =>
  (processingOrder[a.resource?.resourceType] || 999) -
  (processingOrder[b.resource?.resourceType] || 999));
```

Step 3 — A Bundle is just a list (order matters)

IN PLAIN ENGLISH

- A FHIR Bundle wraps everything in `entry[]`; each entry has a `.resource`.
- **We flatten that to a plain array (and still accept a single bare resource).**

`backend/services/fhirService.js`

```
// Accept a whole Bundle OR a single bare resource
const entries = Array.isArray(fhirData.entry)
  ? fhirData.entry           // Bundle -> entry list
  : [{ resource: fhirData }]; // single resource

// Lower number = stored earlier (dependencies first)
const processingOrder = {
  Patient: 1, Organization: 2, Practitioner: 2,
  Observation: 5, Condition: 6, Encounter: 7,
  DiagnosticReport: 8, // needs its Observations first
  Composition: 10,    // references everything -> last
};

// Unknown types sink to the bottom (999)
const sorted = entries.slice().sort((a, b) =>
  (processingOrder[a.resource?.resourceType] || 999) -
  (processingOrder[b.resource?.resourceType] || 999));
```

Step 3 — A Bundle is just a list (order matters)

IN PLAIN ENGLISH

- A FHIR Bundle wraps everything in `entry[]`; each entry has a `.resource`.
- We flatten that to a plain array (and still accept a single bare resource).
- **Resources point at each other: an Observation references its Patient.**

`backend/services/fhirService.js`

```
// Accept a whole Bundle OR a single bare resource
const entries = Array.isArray(fhirData.entry)
  ? fhirData.entry           // Bundle -> entry list
  : [{ resource: fhirData }]; // single resource

// Lower number = stored earlier (dependencies first)
const processingOrder = {
  Patient: 1, Organization: 2, Practitioner: 2,
  Observation: 5, Condition: 6, Encounter: 7,
  DiagnosticReport: 8, // needs its Observations first
  Composition: 10,     // references everything -> last
};

// Unknown types sink to the bottom (999)
const sorted = entries.slice().sort((a, b) =>
  (processingOrder[a.resource?.resourceType] || 999) -
  (processingOrder[b.resource?.resourceType] || 999));
```

Step 3 — A Bundle is just a list (order matters)

IN PLAIN ENGLISH

- A FHIR Bundle wraps everything in `entry[]`; each entry has a `.resource`.
- We flatten that to a plain array (and still accept a single bare resource).
- Resources point at each other: an Observation references its Patient.
- **So we sort by a `processingOrder` map — parents first, the summary Composition last — before saving anything.**

`backend/services/fhirService.js`

```
// Accept a whole Bundle OR a single bare resource
const entries = Array.isArray(fhirData.entry)
  ? fhirData.entry          // Bundle -> entry list
  : [{ resource: fhirData }]; // single resource

// Lower number = stored earlier (dependencies first)
const processingOrder = {
  Patient: 1, Organization: 2, Practitioner: 2,
  Observation: 5, Condition: 6, Encounter: 7,
  DiagnosticReport: 8, // needs its Observations first
  Composition: 10,    // references everything -> last
};

// Unknown types sink to the bottom (999)
const sorted = entries.slice().sort((a, b) =>
  (processingOrder[a.resource?.resourceType] || 999) -
  (processingOrder[b.resource?.resourceType] || 999));
```

Step 4 — Send each resource to the right handler

IN PLAIN ENGLISH

- One switch maps each resourceType to the service that knows how to store it.

backend/services/fhirService.js

```
// Map each resourceType to the service that stores it
function processResource(resource) {
  switch (resource.resourceType) {
    case "Patient":
      return patientService.savePatient(resource);
    case "Observation":
      return observationService.processObservation(resource);
    case "Composition":
      if (resource.type?.coding?.[0]?.code === "34105-7")
        return hospitalDischargeService
          .processHospitalDischarge(resource);
      return null; // other composition types: skip
    // ...one case per supported resource type...
    default:
      console.log("Unhandled:", resource.resourceType);
      return null;
  }
}

// One bad record must NOT stop the whole batch
for (const { resource } of sorted) {
  try { await processResource(resource); }
  catch (err) { console.error(err.message); }
}
```

Step 4 — Send each resource to the right handler

IN PLAIN ENGLISH

- One switch maps each resourceType to the service that knows how to store it.
- **Every resource is wrapped in try/catch — one bad record is logged and skipped, the rest still load.**

backend/services/fhirService.js

```
// Map each resourceType to the service that stores it
function processResource(resource) {
  switch (resource.resourceType) {
    case "Patient":
      return patientService.savePatient(resource);
    case "Observation":
      return observationService.processObservation(resource);
    case "Composition":
      if (resource.type?.coding?.[0]?.code === "34105-7")
        return hospitalDischargeService
          .processHospitalDischarge(resource);
      return null; // other composition types: skip
    // ...one case per supported resource type...
    default:
      console.log("Unhandled:", resource.resourceType);
      return null;
  }
}

// One bad record must NOT stop the whole batch
for (const { resource } of sorted) {
  try { await processResource(resource); }
  catch (err) { console.error(err.message); }
}
```

Step 4 — Send each resource to the right handler

IN PLAIN ENGLISH

- One switch maps each resourceType to the service that knows how to store it.
- Every resource is wrapped in try/catch — one bad record is logged and skipped, the rest still load.
- **Types we don't handle yet are logged, not crashed on.**

backend/services/fhirService.js

```
// Map each resourceType to the service that stores it
function processResource(resource) {
  switch (resource.resourceType) {
    case "Patient":
      return patientService.savePatient(resource);
    case "Observation":
      return observationService.processObservation(resource);
    case "Composition":
      if (resource.type?.coding?.[0]?.code === "34105-7")
        return hospitalDischargeService
          .processHospitalDischarge(resource);
      return null; // other composition types: skip
    // ...one case per supported resource type...
    default:
      console.log("Unhandled:", resource.resourceType);
      return null;
  }
}

// One bad record must NOT stop the whole batch
for (const { resource } of sorted) {
  try { await processResource(resource); }
  catch (err) { console.error(err.message); }
}
```

Step 4 — Send each resource to the right handler

IN PLAIN ENGLISH

- One switch maps each resourceType to the service that knows how to store it.
- Every resource is wrapped in try/catch — one bad record is logged and skipped, the rest still load.
- Types we don't handle yet are logged, not crashed on.
- **A Composition coded 34105-7 is re-routed to the hospital-discharge handler.**

backend/services/fhirService.js

```
// Map each resourceType to the service that stores it
function processResource(resource) {
  switch (resource.resourceType) {
    case "Patient":
      return patientService.savePatient(resource);
    case "Observation":
      return observationService.processObservation(resource);
    case "Composition":
      if (resource.type?.coding?.[0]?.code === "34105-7")
        return hospitalDischargeService
          .processHospitalDischarge(resource);
      return null; // other composition types: skip
    // ...one case per supported resource type...
    default:
      console.log("Unhandled:", resource.resourceType);
      return null;
  }
}

// One bad record must NOT stop the whole batch
for (const { resource } of sorted) {
  try { await processResource(resource); }
  catch (err) { console.error(err.message); }
}
```

Step 5 — One resource, one or more documents

IN PLAIN ENGLISH

- Resolve the reference to find the owner (urn:uuid:... or Patient/123).

backend/services/observationService.js

```
// 1) Who owns this resource? Resolve the reference.
function getIdFromReference(ref) {
  if (!ref) return null;    // no reference -> skip
  return ref.startsWith("urn:uuid:")
    ? ref.slice(9)          // "urn:uuid:ab" -> "ab"
    : ref.split("/").pop(); // "Patient/123" -> "123"
}
const pid = getIdFromReference(obs.subject.reference);

// 2) Make sure the parent Patient exists
let patient = await Patient.findOne({ patientId: pid });
if (!patient)
  patient = await new Patient({ patientId: pid }).save();

// 3) Upsert flat fields + the ORIGINAL (rawFhir)
await Observation.findOneAndUpdate(
  { observationId, testName },
  { $set: { patientId: pid, value, unit, date,
            rawFhir: obs } },
  { upsert: true, new: true });
```

The Mongoose model = one schema per resource type: typed fields for the dashboard plus a rawFhir (Schema.Types.Mixed) field that preserves the complete original resource, so it can be re-exported faithfully.

Step 5 — One resource, one or more documents

IN PLAIN ENGLISH

- Resolve the reference to find the owner (urn:uuid:... or Patient/123).
- **Find-or-create the Patient so children always have a parent.**

backend/services/observationService.js

```
// 1) Who owns this resource? Resolve the reference.
function getIdFromReference(ref) {
  if (!ref) return null;    // no reference -> skip
  return ref.startsWith("urn:uuid:")
    ? ref.slice(9)          // "urn:uuid:ab" -> "ab"
    : ref.split("/").pop(); // "Patient/123" -> "123"
}
const pid = getIdFromReference(obs.subject.reference);

// 2) Make sure the parent Patient exists
let patient = await Patient.findOne({ patientId: pid });
if (!patient)
  patient = await new Patient({ patientId: pid }).save();

// 3) Upsert flat fields + the ORIGINAL (rawFhir)
await Observation.findOneAndUpdate(
  { observationId, testName },
  { $set: { patientId: pid, value, unit, date,
            rawFhir: obs } },
  { upsert: true, new: true });
```

The Mongoose model = one schema per resource type: typed fields for the dashboard plus a rawFhir (Schema.Types.Mixed) field that preserves the complete original resource, so it can be re-exported faithfully.

Step 5 — One resource, one or more documents

IN PLAIN ENGLISH

- Resolve the reference to find the owner (urn:uuid:... or Patient/123).
- Find-or-create the Patient so children always have a parent.
- **Copy the fields the dashboard needs into a flat shape — and keep the whole original in rawFhir.**

backend/services/observationService.js

```
// 1) Who owns this resource? Resolve the reference.
function getIdFromReference(ref) {
  if (!ref) return null;    // no reference -> skip
  return ref.startsWith("urn:uuid:")
    ? ref.slice(9)          // "urn:uuid:ab" -> "ab"
    : ref.split("/").pop(); // "Patient/123" -> "123"
}
const pid = getIdFromReference(obs.subject.reference);

// 2) Make sure the parent Patient exists
let patient = await Patient.findOne({ patientId: pid });
if (!patient)
  patient = await new Patient({ patientId: pid }).save();

// 3) Upsert flat fields + the ORIGINAL (rawFhir)
await Observation.findOneAndUpdate(
  { observationId, testName },
  { $set: { patientId: pid, value, unit, date,
            rawFhir: obs } },
  { upsert: true, new: true });
```

The Mongoose model = one schema per resource type: typed fields for the dashboard plus a rawFhir (Schema.Types.Mixed) field that preserves the complete original resource, so it can be re-exported faithfully.

Step 5 — One resource, one or more documents

IN PLAIN ENGLISH

- Resolve the reference to find the owner (urn:uuid:... or Patient/123).
- Find-or-create the Patient so children always have a parent.
- Copy the fields the dashboard needs into a flat shape — and keep the whole original in rawFhir.
- **upsert makes re-uploading safer: it updates matching records and helps prevent duplicates.**

backend/services/observationService.js

```
// 1) Who owns this resource? Resolve the reference.
function getIdFromReference(ref) {
  if (!ref) return null; // no reference -> skip
  return ref.startsWith("urn:uuid:")
    ? ref.slice(9) // "urn:uuid:ab" -> "ab"
    : ref.split("/").pop(); // "Patient/123" -> "123"
}
const pid = getIdFromReference(obs.subject.reference);

// 2) Make sure the parent Patient exists
let patient = await Patient.findOne({ patientId: pid });
if (!patient)
  patient = await new Patient({ patientId: pid }).save();

// 3) Upsert flat fields + the ORIGINAL (rawFhir)
await Observation.findOneAndUpdate(
  { observationId, testName },
  { $set: { patientId: pid, value, unit, date,
    rawFhir: obs } },
  { upsert: true, new: true });
```

The Mongoose model = one schema per resource type: typed fields for the dashboard plus a rawFhir (Schema.Types.Mixed) field that preserves the complete original resource, so it can be re-exported faithfully.

Missing link — define the model, connect to DB

IN PLAIN ENGLISH

- Every `Patient.findOne / Observation.findOneAndUpdate` call needs a model — here is where it comes from.

backend/models/Observation.js + server.js

```
// ---- define the model ONCE ----  
//      backend/models/Observation.js  
const mongoose = require("mongoose");  
  
const obsSchema = new mongoose.Schema({  
  observationId: String, patientId: String,  
  testName: String, value: Number,  
  unit: String, date: Date,  
  rawFhir: mongoose.Schema.Types.Mixed, // original  
});  
  
module.exports =  
  mongoose.model("Observation", obsSchema);  
  
// ---- connect ONCE at startup (server.js) ----  
mongoose.connect(  
  "mongodb://127.0.0.1:27017/fhir_dashboard");  
// now Observation.findOneAndUpdate(...) just works
```

Repeat the same pattern (a schema + a model) for each resource type you store — Patient, Condition, Procedure... Each is ~10 lines; the rawFhir field is what lets you re-export the original faithfully.

Missing link — define the model, connect to DB

IN PLAIN ENGLISH

- Every Patient.findOne / Observation.findOneAndUpdate call needs a model — here is where it comes from.
- **A Mongoose schema lists the typed fields; mongoose.model() turns it into something you can query.**

backend/models/Observation.js + server.js

```
// ---- define the model ONCE ----  
//      backend/models/Observation.js  
const mongoose = require("mongoose");  
  
const obsSchema = new mongoose.Schema({  
  observationId: String, patientId: String,  
  testName: String, value: Number,  
  unit: String, date: Date,  
  rawFhir: mongoose.Schema.Types.Mixed, // original  
});  
  
module.exports =  
  mongoose.model("Observation", obsSchema);  
  
// ---- connect ONCE at startup (server.js) ----  
mongoose.connect(  
  "mongodb://127.0.0.1:27017/fhir_dashboard");  
// now Observation.findOneAndUpdate(...) just works
```

Repeat the same pattern (a schema + a model) for each resource type you store — Patient, Condition, Procedure... Each is ~10 lines; the rawFhir field is what lets you re-export the original faithfully.

Missing link — define the model, connect to DB

IN PLAIN ENGLISH

- Every Patient.findOne / Observation.findOneAndUpdate call needs a model — here is where it comes from.
- A Mongoose schema lists the typed fields; mongoose.model() turns it into something you can query.
- **rawFhir: Mixed stores the untouched original next to the flat fields.**

backend/models/Observation.js + server.js

```
// ---- define the model ONCE ----  
//      backend/models/Observation.js  
const mongoose = require("mongoose");  
  
const obsSchema = new mongoose.Schema({  
  observationId: String, patientId: String,  
  testName: String, value: Number,  
  unit: String, date: Date,  
  rawFhir: mongoose.Schema.Types.Mixed, // original  
});  
  
module.exports =  
  mongoose.model("Observation", obsSchema);  
  
// ---- connect ONCE at startup (server.js) ----  
mongoose.connect(  
  "mongodb://127.0.0.1:27017/fhir_dashboard");  
// now Observation.findOneAndUpdate(...) just works
```

Repeat the same pattern (a schema + a model) for each resource type you store — Patient, Condition, Procedure... Each is ~10 lines; the rawFhir field is what lets you re-export the original faithfully.

Missing link — define the model, connect to DB

IN PLAIN ENGLISH

- Every Patient.findOne / Observation.findOneAndUpdate call needs a model — here is where it comes from.
- A Mongoose schema lists the typed fields; mongoose.model() turns it into something you can query.
- rawFhir: Mixed stores the untouched original next to the flat fields.
- **One connect() call at startup wires Node to MongoDB — then every model just works.**

backend/models/Observation.js + server.js

```
// ---- define the model ONCE ----  
//      backend/models/Observation.js  
const mongoose = require("mongoose");  
  
const obsSchema = new mongoose.Schema({  
  observationId: String, patientId: String,  
  testName: String, value: Number,  
  unit: String, date: Date,  
  rawFhir: mongoose.Schema.Types.Mixed, // original  
});  
  
module.exports =  
  mongoose.model("Observation", obsSchema);  
  
// ---- connect ONCE at startup (server.js) ----  
mongoose.connect(  
  "mongodb://127.0.0.1:27017/fhir_dashboard");  
// now Observation.findOneAndUpdate(...) just works
```

Repeat the same pattern (a schema + a model) for each resource type you store — Patient, Condition, Procedure... Each is ~10 lines; the rawFhir field is what lets you re-export the original faithfully.

What you learned

The whole pipeline, in six moves

- Read the file in the browser with `file.text()` — no multipart upload or `FileReader` needed.
- POST the raw JSON; parse it on the server with `express.json()` (raise the limit).
- Flatten the Bundle and sort by dependency order before saving anything.
- Route each `resourceType` to its handler; isolate failures with `try / catch`.
- Store one or more documents per resource — flat fields to query + `rawFhir` for the original.
- Define one Mongoose model per type and connect to MongoDB once at startup.

You can now ingest FHIR data and store it in a MongoDB database.