Step 3: Todo App with REST API

Separation of Concerns with Routes

What Changed?

- Step 1 & 2: All routes in a single `index.js` file
- Step 3: **Separated routes** into modular files
- Added **dedicated REST API** endpoints (`/api/posts`)
- Better organization for scalability and maintainability

Key Improvements

1. **Route Separation** - Web UI routes vs API routes
2. **Modular Design** - Each route file has single responsibility
3. **REST API Standards** - Proper HTTP methods and status codes
4. **Better Error Handling** - Consistent error responses

Directory Structure

```
.
├── index.js
├── routes
│   ├── api.js                    # REST API routes (NEW!)
│   └── router.js                 # Web UI routes
├── util
│   ├── db.js
│   ├── uri.js
│   └── util.js
└── views
    ├── detail.ejs
    ├── edit.ejs
    ├── list.ejs
    ├── nav.ejs
    └── write.ejs
```

New: `routes/` directory with separated concerns

Architecture: Before (Steps 1 & 2)

```
// index.js - Everything in one file
app.get('/', (req, res) => { ... });
app.post('/add', (req, res) => { ... });
app.get('/list', (req, res) => { ... });
app.delete('/delete', (req, res) => { ... });
app.get('/detail/:id', (req, res) => { ... });
// ... 10+ routes in single file
```

Problem: Hard to maintain as app grows

Architecture: After (Step 3)

```
// index.js - Clean and organized
import { createApiRouter } from './routes/api.js';
import { createRouter } from './routes/router.js';

const db = await connect(uri);

app.use('/api', createApiRouter(db));  // NEW: API routes
app.use('/', createRouter(db));        // Web UI routes
```

Benefit: Clear separation, easy to extend

Understanding app.use()

`app.use(path, router)` mounts a router at a specific path prefix.

```
app.use('/api', createApiRouter(db));
app.use('/', createRouter(db));
```

This creates **two separate URL spaces**:

- All API routes are prefixed with `/api`
- All Web UI routes start from `/`

Example with '/api' Prefix

1. Users access web pages like: `/api/posts`
2. From `/api` and `app.use('/api', createApiRouter(db))`, the `createApiRouter` handles these routes.
3. The `router.get('/posts', async (req, res) => { ... });` matches `/api/posts`.
4. This function processes the request and sends back JSON data.

Example with '/' Prefix

1. Users access web pages like: `/list`
2. From `/` and `app.use('/', createRouter(db))`, the `createRouter` handles these routes.
3. The `router.get('/list', async (req, res) => { ... });` matches `/list`.
4. This function processes the request and sends back an HTML page.

URL Structure Example

API Routes (routes/api.js)

```
router.get('/posts', ...)       // → /api/posts
router.get('/posts/:id', ...)   // → /api/posts/:id
router.post('/posts', ...)      // → /api/posts
router.put('/posts/:id', ...)   // → /api/posts/:id
router.delete('/posts/:id', ...)  // → /api/posts/:id
```

Web UI Routes (routes/router.js)

```
router.get('/', ...)              // → /
router.get('/list', ...)          // → /list
router.post('/add', ...)          // → /add
router.get('/detail/:id', ...) // → /detail/:id
```

Why Separate URL Spaces?

1. **Clear Distinction**
    - Users know `/api/*` returns JSON
    - Users know `/*` returns HTML pages
    - No confusion about response format
2. **Different Clients**
    - Web browsers use `/list`, `/detail/:id`
    - Mobile apps, frontend frameworks use `/api/posts`
    - Same data, different interfaces

3. Easy to Apply Different Rules

```
// Only API routes need CORS
app.use('/api', cors());
app.use('/api', apiRouter);

// Only Web UI routes need session
app.use('/', session());
app.use('/', webRouter);
```

4. API Versioning

```
app.use('/api/v1', apiRouterV1);
app.use('/api/v2', apiRouterV2);  // Add new version
app.use('/', webRouter);            // Web stays same
```

Two Types of Routes

Web UI Routes (`router.js`)

- **URL:** `/`, `/list`, `/add`, `/detail/:id`
- **Returns:** HTML pages (EJS templates)
- **Purpose:** Traditional web application
- **Actions:** Redirects, render views

REST API Routes (`api.js`)

- **URL:** `/api/posts`, `/api/posts/:id`
- **Returns:** JSON data
- **Purpose:** For frontend frameworks, mobile apps
- **Actions:** Stateless, machine-readable responses

Side-by-Side Comparison

Web UI: Create a Post

```
// POST /add
router.post('/add', async (req, res) => {
  await runAddPost(req);
  res.redirect('/list');  // Redirect to HTML page
});
```

API: Create a Post

```
// POST /api/posts
router.post('/posts', async (req, res) => {
  const doc = { _id: newId, title, date };
  await posts.insertOne(doc);
  res.status(201).json(doc);  // Return JSON
});
```

REST API Routes - api.js

RESTful Design Principles

- Use HTTP methods properly (GET, POST, PUT, DELETE)
- Return JSON responses
- Use appropriate HTTP status codes
- URL structure: `/api/posts` for collection, `/api/posts/:id` for item

REST API - GET All Posts

```javascript
// GET /api/posts
router.get('/posts', async (req, res) => {
  try {
    const posts = await db.collection(POSTS)
      .find()
      .sort({ _id: 1 })
      .toArray();
    res.json(posts);
  } catch (e) {
    console.error(e);
    res.status(500).json({ error: 'Failed to fetch posts' });
  }
});
```

Returns: JSON array of all posts

REST API - GET Single Post

```
// GET /api/posts/:id
router.get('/posts/:id', async (req, res) => {
  try {
    const id = parseInt(req.params.id, 10);
    if (Number.isNaN(id)) {
      return res.status(400).json({ error: 'Invalid id' });
    }

    const doc = await db.collection(POSTS).findOne({ _id: id });
    if (!doc) {
      return res.status(404).json({ error: 'Not found' });
    }
    res.json(doc);
  } catch (e) {
    res.status(500).json({ error: 'Failed to fetch post' });
  }
});
```

REST API - POST (Create)

```
// POST /api/posts
router.post('/posts', async (req, res) => {
  try {
    const { title, date } = req.body || {};
    if (!title) {
      return res.status(400).json({ error: 'title is required' });
    }

    // Auto-increment ID
    const last = await posts.find({ _id: { $type: 'int' } })
      .sort({ _id: -1 }).limit(1).toArray();
    const newId = last.length ? (last[0]._id + 1) : 1;

    const doc = { _id: newId, title, date };
    await posts.insertOne(doc);
    res.status(201).json(doc);  // 201 Created
  } catch (e) {
    res.status(500).json({ error: 'Failed to create post' });
  }
});
```

REST API - PUT (Update)

```javascript
// PUT /api/posts/:id
router.put('/posts/:id', async (req, res) => {
  try {
    const id = parseInt(req.params.id, 10);
    const { title, date } = req.body || {};

    const update = {};
    if (title !== undefined) update.title = title;
    if (date !== undefined) update.date = date;

    const result = await db.collection(POSTS).findOneAndUpdate(
      { _id: id },
      { $set: update },
      { returnDocument: 'after' }
    );

    if (!result.value) {
      return res.status(404).json({ error: 'Not found' });
    }
    res.json(result.value);
  } catch (e) {
    res.status(500).json({ error: 'Failed to update post' });
  }
});
```

REST API - DELETE

```
// DELETE /api/posts/:id
router.delete('/posts/:id', async (req, res) => {
  try {
    const id = parseInt(req.params.id, 10);
    if (Number.isNaN(id)) {
      return res.status(400).json({ error: 'Invalid id' });
    }

    const result = await db.collection(POSTS).deleteOne({ _id: id });
    if (result.deletedCount === 0) {
      return res.status(404).json({ error: 'Not found' });
    }

    res.json({ ok: true, deletedId: id });
  } catch (e) {
    res.status(500).json({ error: 'Failed to delete post' });
  }
});
```

HTTP Status Codes

- **200 OK** - Successful GET, PUT, DELETE
- **201 Created** - Successful POST (new resource)
- **400 Bad Request** - Invalid input
- **404 Not Found** - Resource doesn't exist
- **500 Internal Server Error** - Server-side error

Using correct status codes helps API consumers!

REST API Benefits

1. **Separation of Concerns** - API logic separate from UI
2. **Reusability** - Can be consumed by any client
3. **Scalability** - Easy to add new endpoints
4. **Testing** - API endpoints easier to test
5. **Multiple Clients** - Web, mobile, desktop apps

Dependency Injection & Factory Pattern

This code shows tight coupling (and is BAD):

```
import { db } from '../util/db.js'; // BAD: direct import
// Instead, use factory function to inject db
```

- Each router gets its own `db` instance
- This code shows the violation of the pattern: Dependency Injection

Solution: Factory Design Pattern

- Injects `db` dependency

```
export function createRouter(db) {
  const router = express.Router();
  // ... setup routes with db
  return router;
}
```

- **Factory function** creates router instance
- Each router is self-contained
- Easy to test and reuse

Testing REST API (E2E Tests)

Use `curl` to test API endpoints:

```
# GET all posts
curl http://localhost:5500/api/posts

# GET single post
curl http://localhost:5500/api/posts/1

# POST new post
curl -X POST http://localhost:5500/api/posts \
  -H "Content-Type: application/json" \
  -d '{"title":"Test","date":"2024-01-01"}'

# PUT update
curl -X PUT http://localhost:5500/api/posts/1 \
  -H "Content-Type: application/json" \
  -d '{"title":"Updated"}'
```

Recommended Testing Approach & Tools

- Start simple and small: curl is great and good enough.
- Make tests automated
- Consider using Postman or Thunder Client support collections.

Running Application

Same as before:

```
npm install
npm start
```

Or with nodemon:

```
nodemon ./index.js
```

Now you have both:

- Web UI at `http://localhost:5500/`
- REST API at `http://localhost:5500/api/posts`

Software Engineering Benefits

1. **Modularity** - Each file has single responsibility
2. **Maintainability** - Easy to find and fix bugs
3. **Scalability** - Add new routes without touching old code
4. **Testability** - Test routes independently
5. **Team Collaboration** - Multiple developers can work on different routes

Key Concepts

- **Router Separation** - Organize routes by purpose
- **URL Prefix with app.use()** - Create distinct URL spaces
- **REST API** - Standardized web service interface
- **Factory Pattern** - Create instances with dependencies
- **Dependency Injection** - Pass dependencies instead of importing
- **HTTP Methods** - GET, POST, PUT, DELETE
- **Status Codes** - Proper error and success responses