End-to-End Express Examples (Updated 2025)

These examples walk through building several small Express apps — from sending text responses, to handling GET/POST variables, to implementing login authentication using PostgreSQL and JWT.

Each example builds on the previous one. All commands assume Node.js 20+ and Express 5.x.

https://github.com/ascott02/end-to-end-317.git

```
git clone https://github.com/ascott02/end-to-end-317.git
```

Overview: What These Examples Teach

Each step demonstrates a key concept in modern backend development:

- **Example 1:** The absolute basics of an Express server.
- **Example 2:** Reading query parameters from URLs.
- **Example 3:** Handling POST form data and using middleware.
- **Example 4:** Connecting Express to a PostgreSQL database and hashing passwords.
- **Example 5:** Protecting routes and sessions with JSON Web Tokens (JWT).

Example 1: Sending Text Back to the Requester

This first example is the simplest possible Express application. It shows how to send a plain text response to an HTTP request.

```
mkdir app_demo
cd app_demo
npm init -y
npm install express
```

app.js

```
import express from 'express';
const app = express();

// This defines a GET route for the root URL ("/")
app.get('/', (req, res) => {
  res.send('Hello, world!'); // Sends a simple response back to the browser
});

// Start listening for HTTP requests on port 3000
app.listen(3000, () => console.log('Server running on port 3000'));
```

Visit http://localhost:3000

This establishes the Express foundation: routing and responding.

Example 2: Accessing GET Variables

This demonstrates how to extract parameters from the query string of a URL — the classic `?key=value` syntax.

```
mkdir get_vars
cd get_vars
npm init -y
npm install express
```

app.js

```
import express from 'express';
const app = express();

// The request object (req) contains a property `query` that holds URL parameters
app.get('/', (req, res) => {
  const name = req.query.name ?? 'stranger'; // Use nullish coalescing to set default
  res.send(`Hello, ${name}!`);
});

app.listen(3000, () => console.log('Listening on port 3000'));
```

Visit http://localhost:3000/?name=andrew

Illustrates how GET requests pass data through URLs.

Example 3: Accessing POST Variables

Now we move to handling POST requests — commonly used for forms or API submissions.

```
mkdir post_vars
cd post_vars
npm init -y
npm install express
```

app.js

`express.urlencoded()` parses data from HTML forms (form-encoded) and populates `req.body` with the parsed object. `express.json()` parses JSON payloads, used by API clients like fetch.

```
import express from 'express';
const app = express();

app.use(express.urlencoded({ extended: true }));
app.use(express.json());
```

```
// Render a simple HTML form when the user visits the root page.
app.get('/', (req, res) => {
  res.send(`<form method="POST">
      <input name="name" placeholder="Enter name" />
      <button type="submit">Submit</button>
    </form>`);
});
// Handle the form submission via POST.
app.post('/', (req, res) => {
  const { name } = req.body; // Data comes from the middleware above
  res.send(name ? `Hello, ${name}` : 'Please supply a name!');
});
// Start the server
app.listen(3000, () => console.log('Server running on port 3000'));
```

Shows how to process HTML form submissions and parse request bodies. Middleware makes Express capable of understanding the data being sent.

Example 4: Registration and Login (PostgreSQL + bcrypt)

- Connect Express to PostgreSQL.
- Securely store user passwords using `bcrypt`.
- Separate logic into routes for modularity.

```
mkdir registration_login
cd registration_login
npm init -y
npm install express pg bcrypt dotenv
mkdir public routes
```

.env

```
PGHOST=localhost
PGUSER=student
PGPASSWORD=student
PGDATABASE=registration_login
PGPORT=5432
```

Database setup (psql)

```
CREATE DATABASE registration_login;
\c registration_login
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(255) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL
);
```

app.js

```
import express from 'express';
import dotenv from 'dotenv';
import pkg from 'pg';
import userRoutes from './routes/users.js';

dotenv.config();
const { Pool } = pkg;
const app = express();
const pool = new Pool();

// Express middleware for JSON and form data parsing
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

```
// Serves static files from the public/ directory
app.use(express.static('public'));

// Make database pool accessible in route handlers
app.set('pool', pool);
`
// Mount routes for user registration and login
app.use('/users', userRoutes);
app.listen(3000, () => console.log('Server running on port 3000'));
```

Sets up backend capable of persisting data and handling auth logic.

routes/users.js

```
import express from 'express';
import bcrypt from 'bcrypt';

const router = express.Router();

// Register new user
router.post('/register', async (req, res) => {
  const { username, password } = req.body;
  try {
    // Hash the password before saving — never store plain text passwords
    const hash = await bcrypt.hash(password, 10);
    const pool = req.app.get('pool');
    await pool.query('INSERT INTO users (username, password) VALUES ($1, $2)', [username, hash]);
    res.json({ message: 'User registered' });
  } catch (err) {
    res.status(500).json({ message: 'Error registering user', error: err.message });
  }
});
```

```
// Authenticate existing user
router.post('/login', async (req, res) => {
  const { username, password } = req.body;
  try {
    const pool = req.app.get('pool');
    const result = await pool.query('SELECT * FROM users WHERE username=$1', [username]);
    if (result.rowCount === 0) return res.status(401).json({ message: 'Invalid credentials' });

    const user = result.rows[0];
    const match = await bcrypt.compare(password, user.password);
    res.json(match ? { message: 'Login successful' } : { message: 'Invalid credentials' });
  } catch (err) {
    res.status(500).json({ message: 'Error logging in', error: err.message });
  }
});

export default router;
```

Introduces password hashing, parameterized queries, and async/await structure.

Example 5: JWT-Protected Login (PostgreSQL)

The final example introduces **token-based authentication**, enabling stateless sessions that scale easily.

Concepts covered:

- Creating and verifying JWTs.
- Using authorization headers.
- Protecting routes on the backend.

```
mkdir jwt_auth
cd jwt_auth
npm init -y
npm install express pg bcrypt jsonwebtoken dotenv
```

.env

```
PGHOST=localhost
PGUSER=student
PGPASSWORD=student
PGDATABASE=registration_login
PGPORT=5432
JWT_SECRET=super_secret_key
```

routes/auth.js

```
import express from 'express';
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
import pkg from 'pg';
import dotenv from 'dotenv';
dotenv.config();
const { Pool } = pkg;
const pool = new Pool();
const router = express.Router();

// Register a new user
router.post('/register', async (req, res) => {
  const { username, password } = req.body;
  try {
    const hash = await bcrypt.hash(password, 10);
    await pool.query('INSERT INTO users (username, password) VALUES ($1, $2)', [username, hash]);
    res.json({ success: true, message: 'User registered successfully' });
  } catch (err) {
    res.status(500).json({ success: false, message: 'Registration error', error: err.message });
  }
});
```

```
// Login route — verifies credentials and returns JWT
router.post('/login', async (req, res) => {
  const { username, password } = req.body;
  try {
    const result = await pool.query('SELECT * FROM users WHERE username=$1', [username]);
    if (result.rowCount === 0) return res.status(401).json({ success: false, message: 'Invalid credentials' });

    const user = result.rows[0];
    const match = await bcrypt.compare(password, user.password);
    if (!match) return res.status(401).json({ success: false, message: 'Invalid credentials' });

    // Create token with 1-hour expiration
    const token = jwt.sign({ id: user.id }, process.env.JWT_SECRET, { expiresIn: '1h' });
    res.json({ success: true, token });
  } catch (err) {
    res.status(500).json({ success: false, message: 'Login error', error: err.message });
  }
});

export default router;
```

This final stage adds authentication tokens that persist across sessions without server-side state.

Using the Token

In your frontend JS:

```
fetch('/api/auth/login', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ username, password })
})
  .then(res => res.json())
  .then(data => {
    if (data.success) localStorage.setItem('token', data.token);
  });
```

Add middleware to verify tokens for protected routes.

```
function verifyToken(req, res, next) {
  const header = req.headers['authorization'];
  if (!header) return res.status(403).json({ message: 'Missing token' });
  const token = header.split(' ')[1];
  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.status(403).json({ message: 'Invalid token' });
    req.user = user; // Attach decoded user info to request
    next(); // Pass control to the next middleware or route
  });
}
```

Demonstrates middleware — reusable logic that intercepts and validates requests before they reach a route handler.

Why JWT?

JWT (JSON Web Token) expands on the classic username-password authentication by letting the server verify users without maintaining session data in memory or a database.

- **Scales easily:** Because the token itself contains the verification data, any server in a cluster can validate it independently. There’s no shared session store to synchronize.
- **Stateless authentication:** The server doesn’t have to remember each logged-in user; it just checks the token’s signature each time.
- **Interoperability:** JWTs are language-agnostic and can be used across different microservices and client types (mobile, web, API).

JWT builds directly on Example 4's logic — users still log in with the database, but now they receive a signed token that must be presented with each request. This adds an additional layer of protection: a *second check* confirming both the user's identity and token validity.

It's complementary to password-based authentication, not a replacement. You still need to validate passwords initially, but JWT adds ongoing verification that scales horizontally across distributed systems.

Summary

- **Example 1:** Basic HTTP server
- **Example 2:** Handling query parameters
- **Example 3:** Parsing POST requests with middleware
- **Example 4:** Integrating PostgreSQL and bcrypt for user management
- **Example 5:** Adding secure, token-based authentication with JWT

Together, these form a miniature evolution of a backend API — from hello world to a protected, production-ready architecture.