Backend & Authorization with Next.js & Express

CIS 1962 (Fall 2025)
November 17th, 2025

Penn Engineering

Lesson Plan

5	**Backend Development**
10	**HTTP & REST API**
20	**Express.js Backend**
34	**Next.js Backend**
41	**Backend Design**
49	**Authorization**
60	**Intro to Databases (Redis)**

Penn Engineering

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673

Penn Engineering

Quick Check!

Final Project Proposal due **_tonight_**

HW4 Extended by a week

HW5 already released

Optional Primsa/ORM crash course during OH (talk to us if interested)

Backend Development
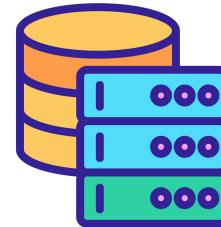
What does it mean to develop the "backend" of a web application?

What is a Backend?

The **backend** of a web application is the logic that needs to happen behind the scenes. This includes tasks like:

Security & Authorization

Data Storage & Database Access

Business Logic: Rules, Calculation, Validation

Full-Stack Illustration

Frontend

Backend

```
await fetch('/api/fetchjoke');
```

CLICK BELOW FOR A JOKE!

GET JOKE

WHY DID DRACULA LIE IN THE WRONG COFFIN? HE MADE A GRAVE MISTAKE.

JavaScript & Backend

With Node.js, you can write code to run on a server, because it is a runtime environment that allows networking and file system access.

While you can create a backend manually with Node, there are many frameworks that make it easier:

- Express.js
- Next.js

```javascript
// Without Next or Express
const http = require('http');

const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'application/json');

  if (req.method === 'GET' && req.url === '/') {
    res.statusCode = 200;
    res.end(JSON.stringify({ message: 'Hello World!' }));
  } else {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: 'Not found' }));
  }
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

General Backend Terms

API: Application programming interface, or how the frontend will talk to the backend

Endpoint: The specific URL the frontend (or terminal) can call or get/send data

Request/Response: The flow of data to (request) and from (response) the backend

Business Logic: The rules, calculations, and flow of an application handled by the backend (often in a service layer)

HTTP & REST API

What does it mean to develop the "backend" of a web application?

HTTP

HTTP, or Hypertext Transfer Protocol, is the foundational protocol of the web. It allows a client like a browser or a developer in a terminal to communicate with servers.

You work with HTTP through **requests** and **responses**.

Penn Engineering

HTTP Requests

```
curl -X POST http://localhost:3000/login \
     -H "Content-Type: application/json" \
     -d '{"user": "voravich"}'
```

An HTTP request needs to know certain things:

- **Method**: Verbs describing what actions you want to perform with the request, including GET, POST, PUT, PATCH, and DELETE.
- **URL**: Where you want to make a request
- **Headers**: important authentication information or specifying data types
- **Body**: Used for POST/PUT to send data

Penn Engineering

HTTP Verbs

These are some best practices when using HTTP verbs:

- GET: used to **request data**, should never change the server's application state.
- DELETE: used to **request the removal of an item** on the server.

GET/DELETE
Request
something

- POST: used to **create a new object** on the server
- PUT: used to **replace an existing object** with new data
- PATCH: used to **modify an existing object** with new data
  - PUT/PATCH should be idempotent: two of the same request should put the server in the same state for both.

POST/PUT/PATCH
Send data

Penn Engineering

13

Fetch and curl Examples

curl (cURL): command to be used to make an HTTP request

```
curl "https://hw4.cis1962.esinx.net/api/pokemon/pikachu"
```

Fetch API: JS interface for making HTTP requests and working with responses

```js
async function getJoke() {
    const res = await fetch('https://official-joke-api.appspot.com/random_joke');
    const data = await res.json();
    return data;
}

getJoke().then((data) => console.log(data));
```

Penn Engineering

HTTP Responses

HTTP Responses output the following items:

- **Status Code**: Indicate whether HTTP request was successful, and various other <u>states</u>. (200 = OK, 404 = Not Found, 500 = Error, etc)
- **Headers**: Information about the response, like "Content-Type: application/json"
- **Body**: Often JSON with requested data

HTTP Status Codes

200 OK – Request succeeded

201 Created – Resource created

204 No Content – Successful, no response body

301 Moved Permanently – Resource has a new permanent URL

302 Found – Temporary redirect

400 Bad Request – Client request error

401 Unauthorized – Authentication required

403 Forbidden – Request refused

404 Not Found – Resource doesn't exist

500 Internal Server Error – Server encountered an error

503 Service Unavailable – Server temporarily unavailable

Penn Engineering

REST API

Backends will often use an architectural design style called **REST (R**epresentational **S**tate **T**ransfer**)**.

Using REST gives structure and conventions to using HTTP, making them predictable across many systems.

REST maps HTTP methods to CRUD (create, read, update, delete) operations to standardize how resources are managed on the web.

REST Example: API Routes

Imagine we have an backend to manage users.

Here are a few URL names in our API to show REST conventions:

- GET /users – List users
- GET /users/9057 – Fetch user with ID 9057
- POST /users – Create new user
- PUT/PATCH /users/9057 – Update user 9057
- DELETE /users/42 – Delete user 9057

HTTP & REST Overview

HTTP provides the methods,
REST provides the rules.
Together they allow for structured communication on the web.

Now, how do we write JS backend code using these ideas?

Express.js Backend

How do we use Express.js to build a backend web server and API routes?

Express.js

Express.js is the most popular web framework for Node.

It lets you build web servers and APIs easily within JS, with the following features:

- **Flexible, simple routing**: Easily map URLs and HTTP methods
- **Middleware**: Able to handle authentication, logging, and JSON with "plug-in" methods
- **Well-supported:** has a large ecosystem of compatible libraries and can run wherever Node.js does.

Creating an Express App

[p]npm i express
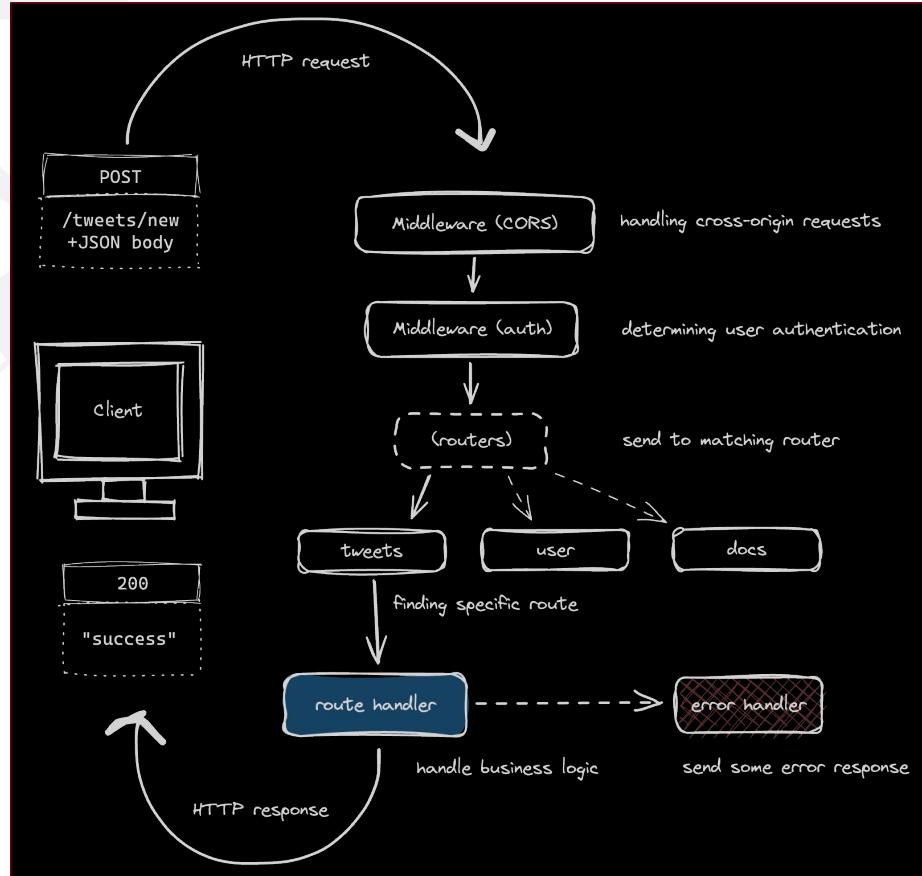
Once you import express into a script, you can create an app and let it listen to a certain port (often port 3000). This starts your Express server within the current terminal session.

```javascript
// app.js
const express = require('express');
const app = express();

// Routes and Middleware
...

app.listen(3000, () => console.log('Server running'));
```

Anatomy of Express.js

https://imgur.com/D7SxSjT

1. HTTP Request
2. Middleware
3. Routers
4. Route/Error Handlers
5. HTTP Response

Penn Engineering

23

Express Routing Syntax

A route in Express defines in the server the HTTP method and URL path you want to handle, and what code to run for those requests.

Syntax:

```
app.METHOD(PATH, HANDLER)
```

METHOD: HTTP verb like `get, post, put, delete`

PATH: endpoint URL

HANDLER: function to run when route is matched

Express.js Syntax Example

```javascript
const express = require('express');
const app = express();

app.get('/api', (req, res) => {
  res.send('Hello, Express!');
});

app.post('/api/data', (req, res) => {
  // Send the data somewhere first
  res.json({ received: req.body });
});

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

`app.get()` at route '/api'
- GET request
- res.send() can send a response of any type, automatically setting the `Content-Type` header.

`app.post()` at route '/api/data'
- POST request
- res.json() specifically sends a JSON, using `Content-Type: application/json`

Modularizing with express.Router()

A router object in Express can help modularize your backend code by defining subsections that only run on certain paths.

Think of it as a "mini Express app" that can group related routes together.

```javascript
// userRoutes.js
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.send('List of users');
});
...

module.exports = router;
```

```javascript
// app.js
const express = require('express');
const userRoutes = require('./userRoutes');
const app = express();

app.use('/users', userRoutes);

app.listen(3000, () => console.log('Server running'));
```

Middleware

Middleware functions run before any router handlers.

Common uses for middleware include logging, authentication, validation, and error handling.

You declare middleware with `app.use()` for global (before every request) middleware, or defined as a function for specific routes.

```
app.use((req, res, next) => {
  console.log('I am global!');
  next();
});


app.get('/hello', (req, res) => {
  res.send('Hello World!');
});
```

```
function requireAuth(req, res, next) {
  ...
  next();
}

app.get('/dashboard', requireAuth, (req, res) => {
  res.send('Welcome to the dashboard!');
});
```

Middleware Chaining

You can chain multiple middleware functions in the same app.use() call, or attach multiple middleware functions to routes. These will be called one after another with the next() method.

```js
const setFoo = (req, res, next) => {
  req.foo = 'some value';
  next();
};

const logFoo = (req, res, next) => {
  const { foo } = req;
  console.log(foo);
  next();
};

app.use(setFoo, logFoo);
```

```js
function requireAuth(req, res, next) {
  ...
  next();
}

function logAccess(req, res, next) {
  console.log(`Accessed: ${req.path}`);
  next();
}

app.get('/settings', requireAuth, logAccess, (req, res) => {
  res.send('Settings Page');
});
```

Error Handling

Sometimes errors occur in your routes. You can pass an error argument to `next()` so that error handling middleware can handle it.

```javascript
app.get('/findUser', (req, res, next) => {
  ...
  if (user) {
    res.json(user)
  } else {
    next(new Error('Oh no! Something broke.'));
  }
});
```

Error Handling Middleware

Error-handling middleware has one difference: it includes an `err` argument.

Any error handling middleware should be written **after** all other middleware in an app.

```js
app.get('/fail', (req, res, next) => {
  next(new Error('Oh no! Something broke.'));
});

app.use((err, req, res, next) => {
  if (err.message === "Oh no! Something broke.") {
    return res.status(400).json({ error: 'Specific fail.' });
  }
  next(err);
});

app.use((err, req, res, next) => {
  res.status(500).json({ error: 'Something went wrong.' });
});
```

Penn Engineering

Parameters & Queries

Route Parameters (`:param`): parts of a URL act as variables

```javascript
app.get('/users/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
// GET /users/17 => 'User ID: 17'
```

Query Strings (`/?key=value`): key-value pairs that appear after a ? in the url

```javascript
app.get('/search', (req, res) => {
  const searchQuery = req.query.q;
  const pageNum = req.query.page;
  res.send(`Results for: ${searchQuery}, page: ${pageNum}`);
});
// GET /search?q=express&page=2 => 'Results for: express, page: 2'
```

Server-side Navigation

For multi-page applications (NOT SPAs like React), you can use express to perform server-side routing. This allows rendering of different HTML files to different routes.

```javascript
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'views', 'home.html'));
});

app.get('/about', (req, res) => {
  res.sendFile(path.join(__dirname, 'views', 'about.html'));
});

app.get('/contact', (req, res) => {
  res.sendFile(path.join(__dirname, 'views', 'contact.html'));
});
```

Penn Engineering

Note: Express & SPAs

Since SPAs often need to serve up a single index.html and static assets, Express is not sufficient for navigation in SPAs.

For instance, in React, Express would need to serve up the React app for all non-API routes (made through React Router) using the following syntax:

```
app.use(express.static(path.join(__dirname, 'dist'))); // static assets

app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'dist', 'index.html'));
});
```

Next.js Backend

How do we build a backend with Next.js?

App Router & route.ts

Next.js's App Router provides us ways to define components, layouts, and API endpoints through the file system in the `/app` folder.

One of these files is the `route` file, which defines an API endpoint complete with support for HTTP method handlers that allow server functions to run.

HTTP Methods in route.ts

In `route.ts`, you can export specific functions for each HTTP method. For instance:

```typescript
// api/helloworld/route.ts
export async function GET() {
  return Response.json({ message: 'Hello World!' });
}


export async function POST(request) {
  const data = await request.json();
  // ...do something with data
  return Response.json({ received: data });
}
```

Route Parameters

We can access route parameters with Next.js's dynamic routes (ones with []) similar to how we did with normal routes:

```ts
// app/api/users/[id]/route.ts
export async function GET(request: Request, { params }: { params: { id: string } }) {
  const userId = params.id;
  return Response.json({ userId });
}
```

Query Strings

We can also access query strings by accessing the URL object associated with the current URL:

```typescript
export async function GET(request: Request) {
  const { searchParams } = new URL(request.url);

  const role = searchParams.get('role');
  const page = searchParams.get('page');

  return Response.json({ role, page });
}
```

Middleware with proxy.ts

Within the Next.js backend, you can use `proxy.ts` to handle middleware, any code that runs before incoming requests.

```ts
export function proxy(request: Request) {
  if (!isAuthenticated(request)) {
    return Response.json(
      { success: false, message: 'authentication failed' },
      { status: 401 }
    )
  }
}
```

Penn Engineering

5-Minute Break!

Penn Engineering

Backend Design

How do we design a backend well for maintainability, scalability, and efficiency with data handling?

Good Backend Design

A well-designed backend is made to be scalable, maintainable, secure, and performant.

It will be beneficial to use a proper backend design pattern that's been adopted by many teams.

Backend Layers

A popular pattern for organizing code on the backend involves splitting backend code into 2-3 layers:

- **Controller Layer**: handles HTTP requests and responses
- **Service Layer**: Business logic, transform data, enforce rules and validation
- **Data Access Object (DAO)**: Directly manage data storage mechanism, like through the file system or database.

This separation keeps apps maintainable and scalable.

Router

When dealing with complex backends, it's beneficial to define Routers so that you can organize routes.

This lets you apply middleware or controllers in a clean way.

```js
// app.js
app.use('/user', userRoutes);
```

```js
// userRoutes.js
import { Router } from 'express';
import { getUser, createUser } from
'../controllers/userController.js';
import { authenticateToken } from '../auth.js';

const router = Router();

router.use(authenticateToken);
router.get('/', getUser);
router.post('/', createUser);

export default router;
```

Controller Layer

The **controller** is responsible for receiving HTTP requests to send to the service layer, and outputting responses.

This is one of the first points of contact for the backend, where you define how to deal with requests and responses.

```javascript
import { getUserDataService } from '../services/userService.js';

export async function getUser(req, res) {
  const user = req.user;
  const data = await getUserDataService(user);
  res.json({ msg: `Got user: ${user}`, data });
}
...
```

Service Layer

The **service layer** handles any business logic. This should include either functions to directly fetch data, or to call data access/DAO methods to acquire the data.

You may also handle logic like validation and data cleaning here, anything to transform the contents of the request or response.

```javascript
import { getUserData, setUserData } from '../DAO/userDAO.js';

export async function getUserDataService(user) {
  return getUserData(user);
}

export async function setUserDataService(user, data) {
  return setUserData(user, data);
}
```

DAO Layer

DAO, or a Data Access Object, separates the direct database or data access code from the business logic.
While DAO is often optional for the pattern, it is beneficial to encapsulate the access to the data source for testing.

This makes it easier to say, swap databases easily.
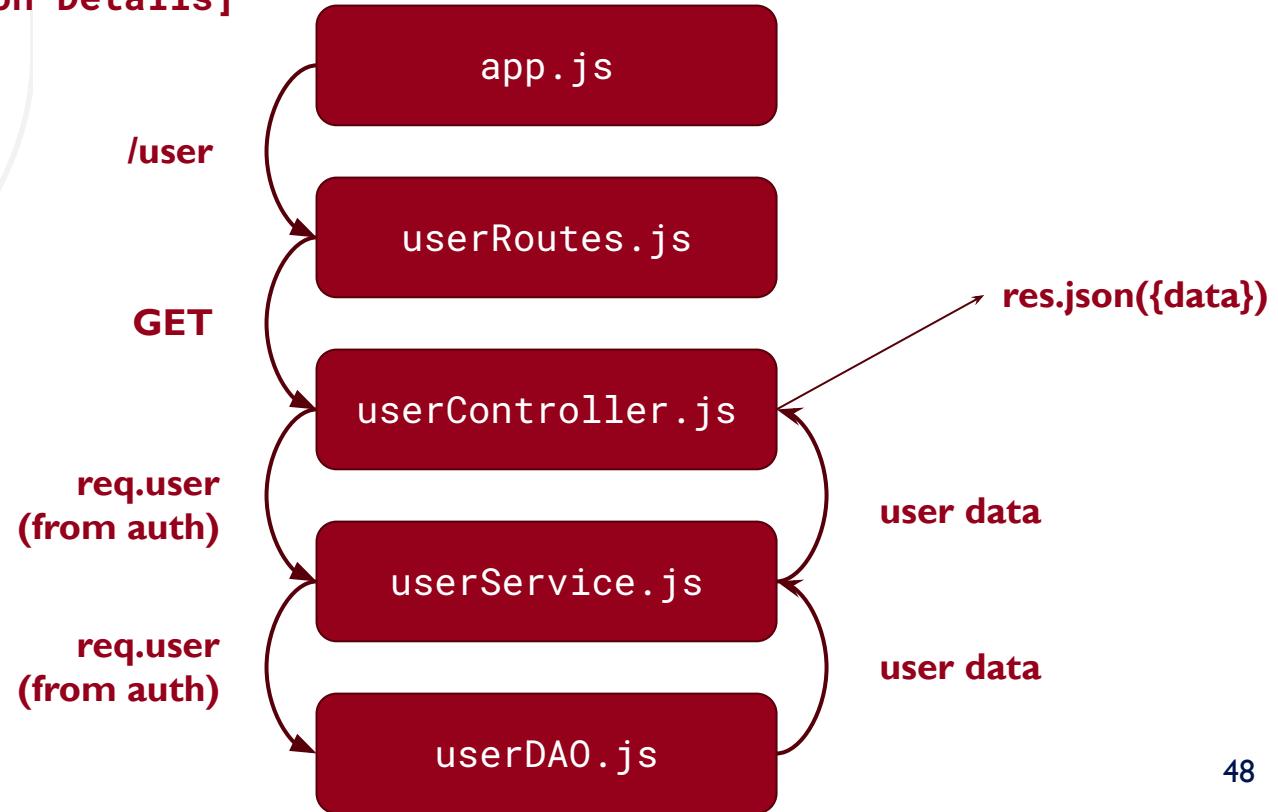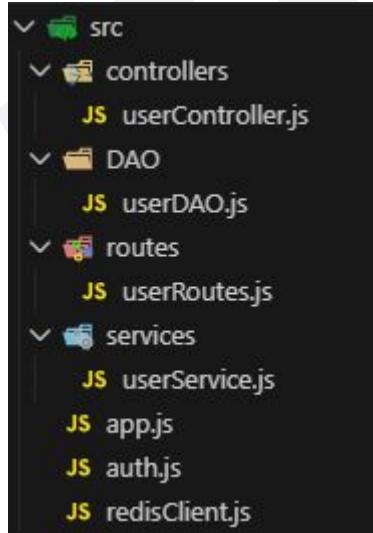
```javascript
import redisClient from '../redisClient.js';

export async function getUserData(user) {
  const str = await
redisClient.get(`${user}:data`);
  return str ? JSON.parse(str) : null;
}

export async function setUserData(user,
data) {
  await redisClient.set(`${user}:data`,
JSON.stringify(data));
}
```

Backend Illustration

```
curl http://localhost:3000/user
        -H [Authorization Details]
```



```
> src
  > controllers
      JS userController.js
  > DAO
      JS userDAO.js
  > routes
      JS userRoutes.js
  > services
      JS userService.js
  JS app.js
  JS auth.js
  JS redisClient.js
```

app.js

/user

userRoutes.js

GET   res.json({data})

userController.js

**req.user
(from auth)**   **user data**

userService.js

**req.user
(from auth)**   **user data**

userDAO.js

Penn Engineering

48

Authorization

How do we use authorization such as JWT to secure our routes?

Authentication vs Authorization

Authentication: Who are you?

- Example: Login

Authorization: What are you allowed to do?

- Example: Accessing specific resources, admin privileges, dashboards only logged-in users can see

Authorization in APIs

We want to be able to protect certain routes by only allowing access to certain people with permissions.

We can enforce these permissions by writing authorization middleware that requires certain tokens with permission.

JWT Authorization

JWT (JSON Web Token) is a compact way to transmit info & permissions as a signed token.

Each token has the following structure:

HEADER.**PAYLOAD**.**SIGNATURE**

For instance:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoidm9yYXZpY2giLCJpYXQiOjE3NjMyNz
kxMjksImV4cCI6MTc2MzM2NTUyOX0.y58QBdd7THJCf7CupPp4Ca11gtdRp_7iik6u9TzUX5k
```

Anatomy of a JWT Token

HEADER.PAYLOAD.SIGNATURE

The **header** describes the token itself, including type (“`typ`”: “JWT”), and signing algorithm (`e.g. "alg": "HS256"`)

The **payload** contains the data, the user identity and permissions, that you want to transmit.

The **signature** ensures data integrity (that the info hasn't been tampered with) by encrypting the header and payload with a secret.

Penn Engineering
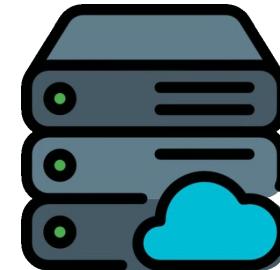
Example JWT Flow

1. User logs in, backend validates credentials against database/user storage

Example JWT Flow

2. Server generates a JWT token with user's identity and roles. This involves attaching a payload, a secret key (for encryption), and other options, like an expiration time.

```
const JWT_SECRET = process.env.JWT_SECRET || 'my_secret_dont_steal_pls';

jwt.sign({ user }, JWT_SECRET, { expiresIn: '24h' });
```
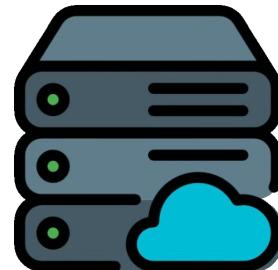
Example JWT Flow

3. User should now have the JWT token, which can be used in future requests that need it (Authorization header)

"Can I
GET /api/box/clx1234567890?
-H [Insert Key Here]"

"Sure, one moment…"

Example JWT Flow
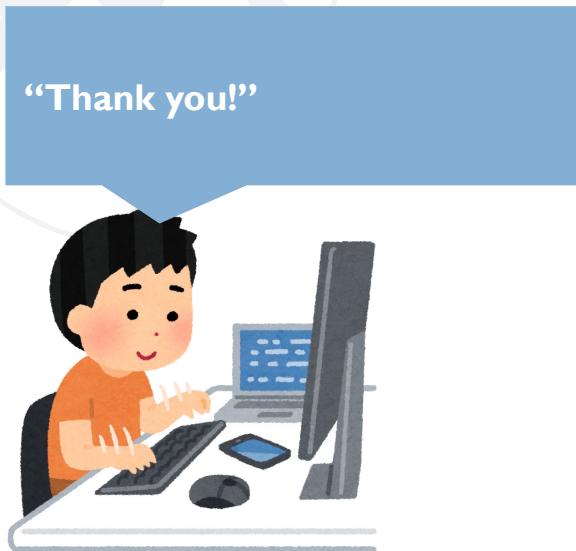
4. The server will authenticate the user by verifying the token and the permissions granted to the user.

```javascript
jwt.verify(token, JWT_SECRET, (err, payload) => {
    … // Check errors before returning
    req.user = payload.user; // use the payload
    next(); // this is middleware, move to the next one!
}
```

Example JWT Flow
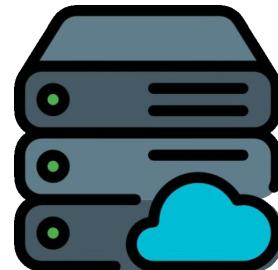
5. The server allows or denies the action. If allowed, the action is performed.

"Thank you!"

"Here is your requested response:"

The .env file

The .env file stores environment variables in your application, so that you can keep your sensitive configurations and secrets out of your codebase. Items like:

- Database Credentials
- API/Secret keys
- App/Environment settings

```
PORT=3000
JWT_SECRET=supersecret
REDIS_URL=redis://localhost:6379
```

The file is loaded on app startup, often through libraries like `dotenv`, into the app's `process.env`.

You should **not** commit a `.env` file to a git repo. (use `.env.example` as a template)

Penn Engineering

Intro to Databases

What is a database and how do they integrate into our backend development?

What is a Database?

A database is an organized collection of data that is stored and accessed electronically.

They allow for storing, querying, updating, and securing data.

In a backend, you will use a client/driver to connect to some database using some credentials.

- `pg/postgres` for postgres
- `mongoose` for MongoDB
- `redis` for Redis

Traditional vs. Cloud DBs

Traditional Databases

- Self-hosted, requires hosting on a local machine, data center, or virtual machine.
- You are responsible for setup, installation, updates, etc.

Cloud Databases

- Managed by cloud vendors (AWS, Google)
- Runs on their infrastructure, just requires your credentials
- Pay per use

Database Types

Relational (SQL)

- Tables, schemas, relationships (e.g., PostgreSQL, MySQL)

NoSQL (Non-relational)

- Key-Value Stores (e.g., Redis)
- Document Stores (e.g., MongoDB)
- Wide Column, Graph, Time-Series, etc.

Each type optimized for different use-cases.

Penn Engineering

Redis

Redis (**Re**mote **Di**ctionary **S**erver) is a NoSQL key-value store database.

It is extremely fast due to keeping data in RAM (both for local and for cloud versions!)

It's useful for tasks like caching, session management, and real-time analytics that require fast access.

Setting up Redis (Local)

First, install Redis locally (or use a portable release like <u>this</u>).

Then, install an npm dependency: `npm i redis`

Start the Redis server locally, which should run on
`localhost:6379`

The following code will create a client for you to use:

```
const client = createClient();
client.connect().catch(console.error);
client.on('error', err => console.log('Redis Client Error', err));
```

Setting up Redis (Cloud)

Create a Redis Cloud account (https://redis.io/cloud/)

Create a cloud database on their servers

Connect to the database endpoint in your code:

```javascript
const client = createClient({
    username: '...',
    password: '...',
    socket: {
        host: '...',
        port: ...
    }
});

client.connect().catch(console.error);
client.on('error', err => console.log('Redis Client Error', err));
```

Using Redis

Once you have your client, you can use the get/set methods achieve data persistence.

Common convention is to use colon (:) separators to organize, define namespaces, and avoid collisions.

```javascript
import redisClient from '../redisClient.js';

export async function getUserData(user) {
  const str = await redisClient.get(`${user}:data`);
  return str ? JSON.parse(str) : null;
}

export async function setUserData(user, data) {
  await redisClient.set(`${user}:data`, JSON.stringify(data));
}
```

Demo: Putting it All Together

Let's demonstrate an Express.js backend with authorization using a Redis database.

We will need to make a cloud database (https://redis.io/cloud/) or a local database on Redis to use this code.

To use JWT, you will need to install it: `npm i jsonwebtoken`