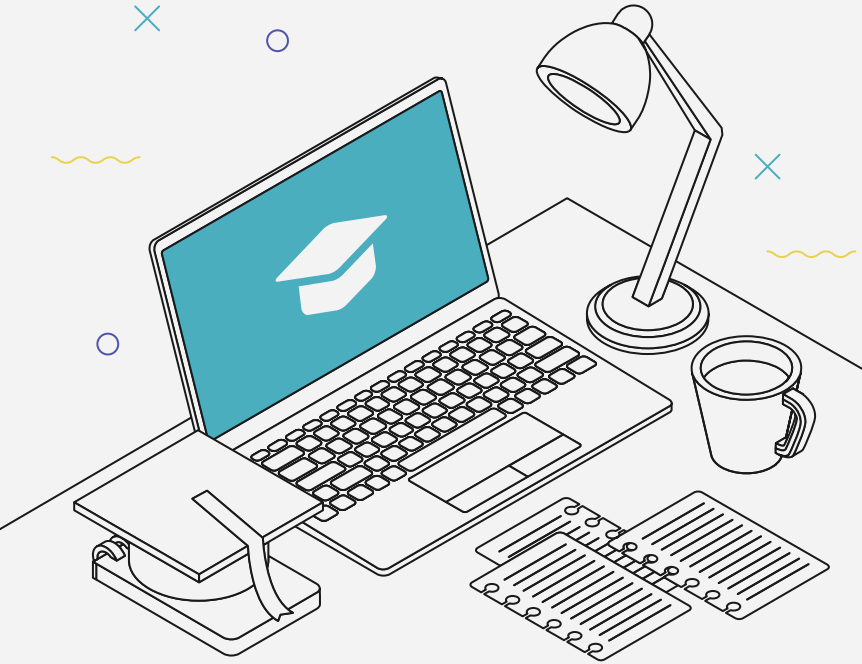
Github

https://github.com/PersoSirEduard/HackMcGill-Backend-Workshop

Backend Workshop

HackMcGill



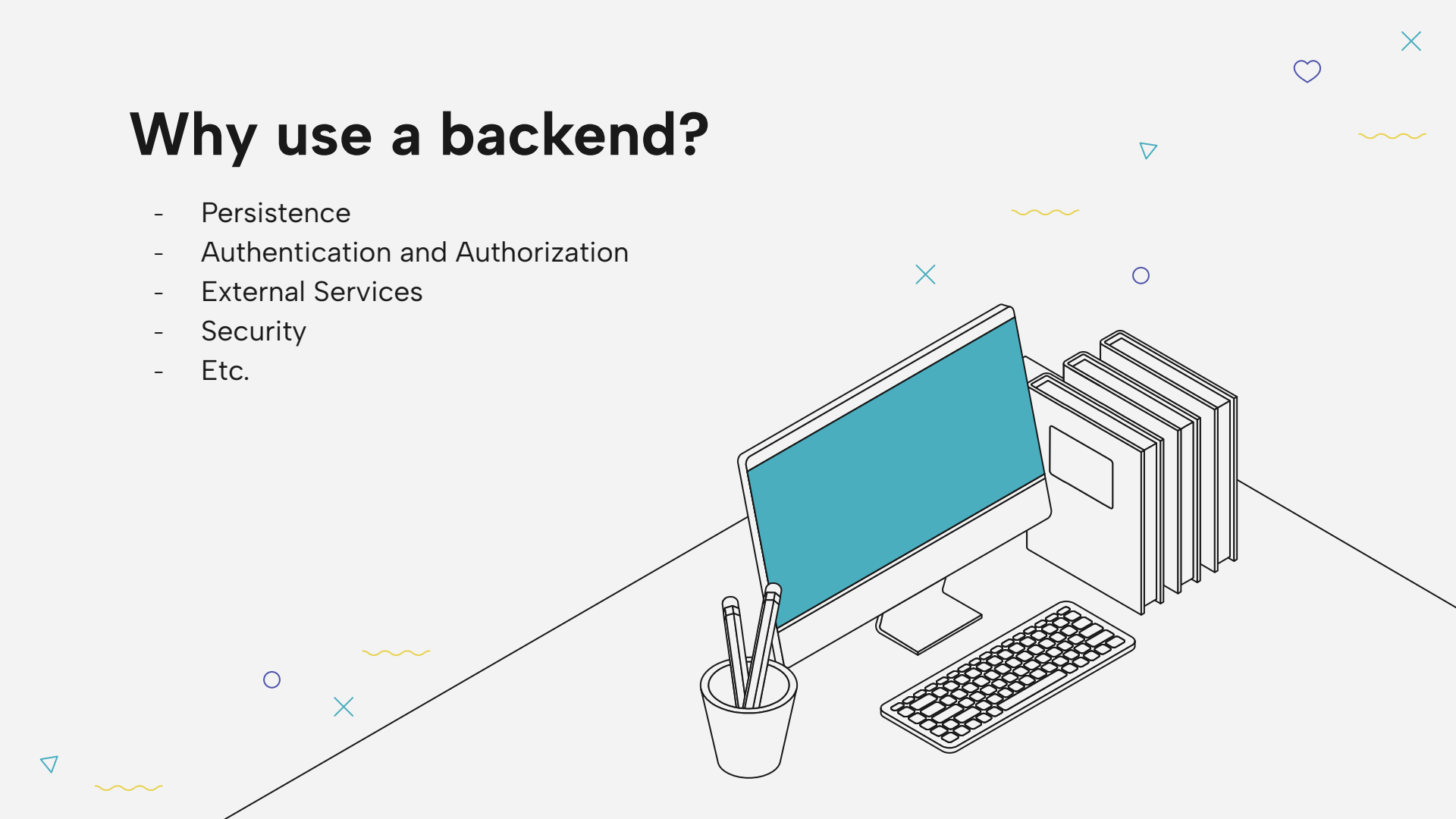
Eduard Anton

What is a backend?

The backend (or “server side”) is the portion of the website you don’t see. It’s responsible for storing, [processing] and organizing data, and ensuring everything on the client-side actually works. The backend communicates with the frontend, sending and receiving information to be displayed as a web page. Source: CareerFoundry

Why use a backend?

- Persistence
- Authentication and Authorization
- External Services
- Security
- Etc.

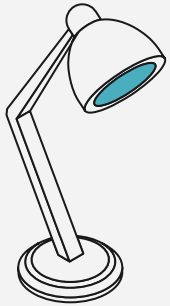


Application Programming Interface (API)

- Allows software to communicate
- Specify standards (interface)
- Software design involved
- **APIs are not necessarily provided by a remote server! (e.g. Windows API)**

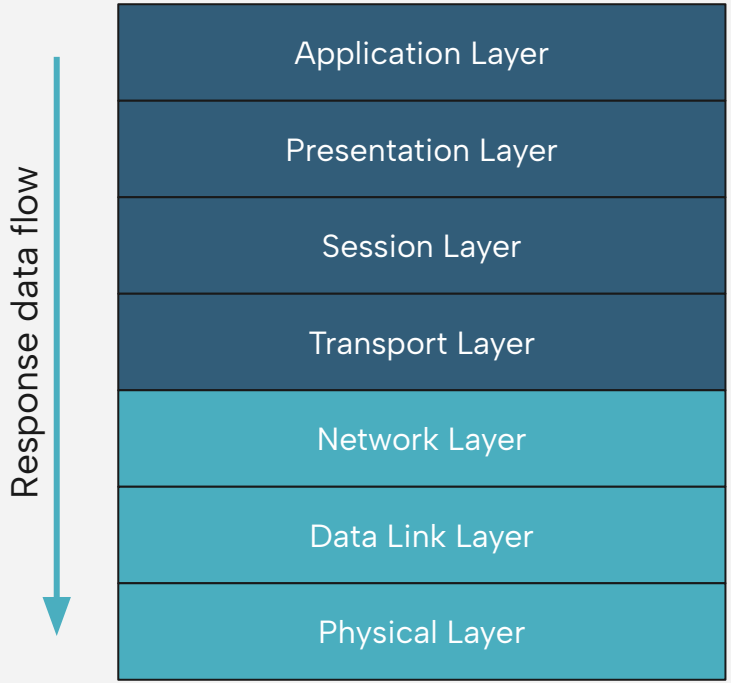
Open Systems Interconnection (OSI)

- Standardization of communication (AKA protocols)
- Communication over the network
- **Layered abstraction**



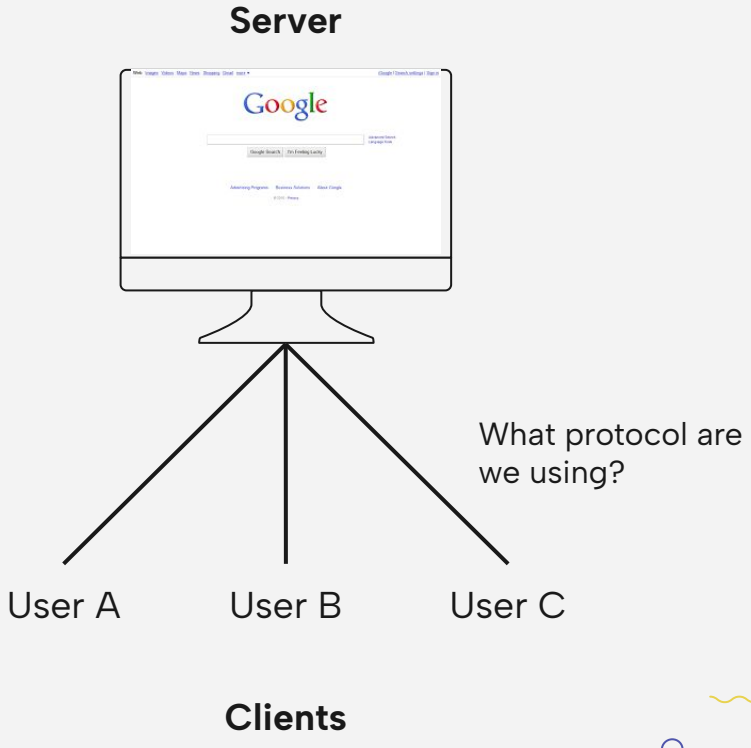
From the **Application Layer** and up we care about:

- The message we want to send
- The destination of our message

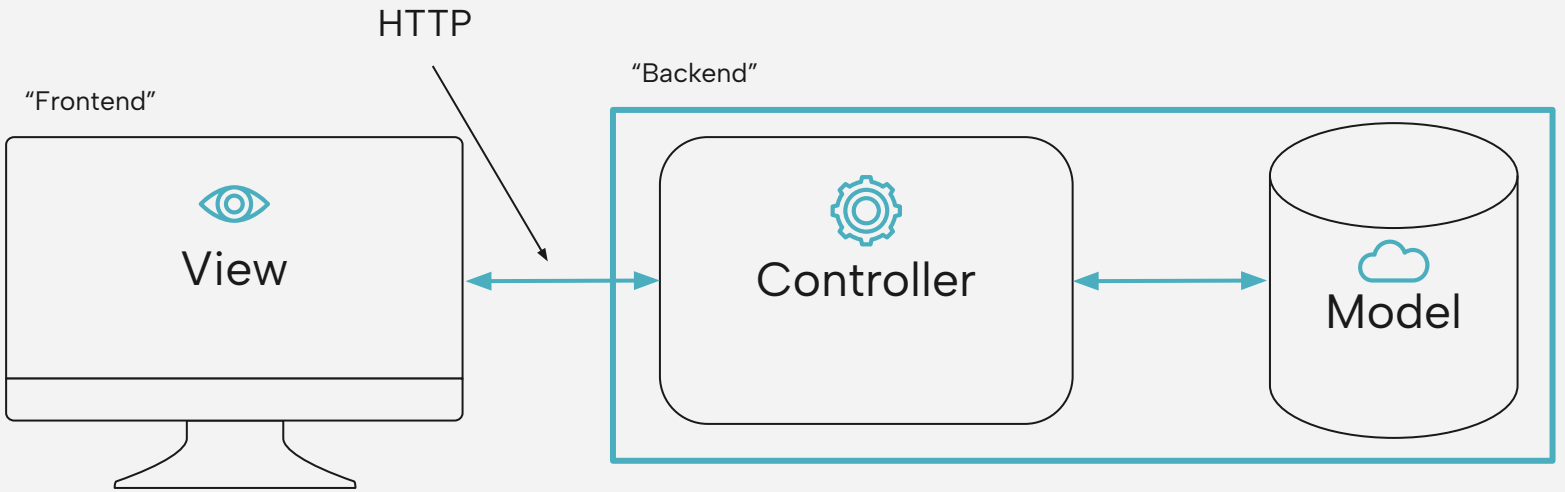


Client-Server

- Client requests a “service” provided by the server
- Agreement on the API
- Centralized architecture
- Simple
- Many-to-one connections
- Careful with traffic management. There is a risk of Denial-of-service (DoS) attacks
- Other architectures: P2P



MVC (Model-View-Controller)



Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes, such as machine-to-machine communication, programmatic access to APIs, and more.

Source: Mozilla

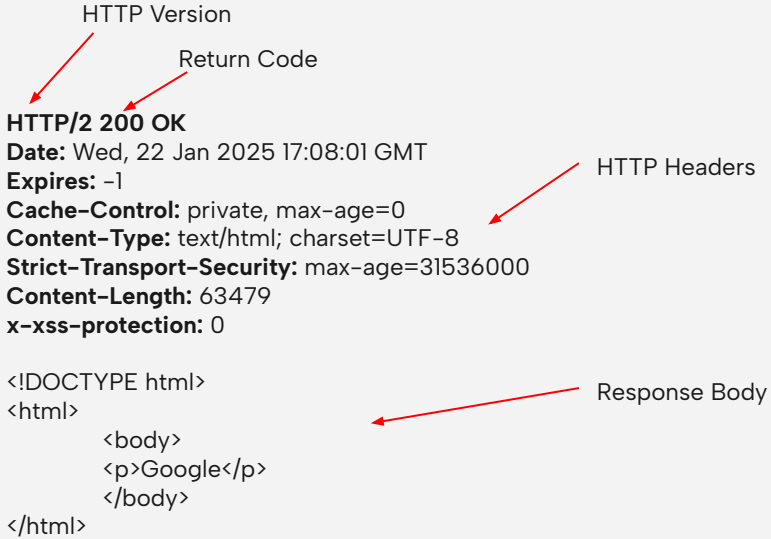
```
<!DOCTYPE html>
<html>
        <head>
        <title>Example</title>
        </head>
        <body>
        <p>This is an example of a simple HTML
page with one paragraph.</p>
        </body>
</html>
```

Note: HTTP by itself is not our API

HTTP Requests and Responses

Request

Response



HTTP Versions

1.0

- First version, introduced in 1996
- Stateless (no sessions)

1.1

- Improved performance and security
- Persistent connection
- Caching

1.2

- Pipelining (send multiple requests before a response)
- Server push (proactive server)
- Header compression

2.0

- Prioritization
- Streams
- Compression

Let’s build our API!

Express.js

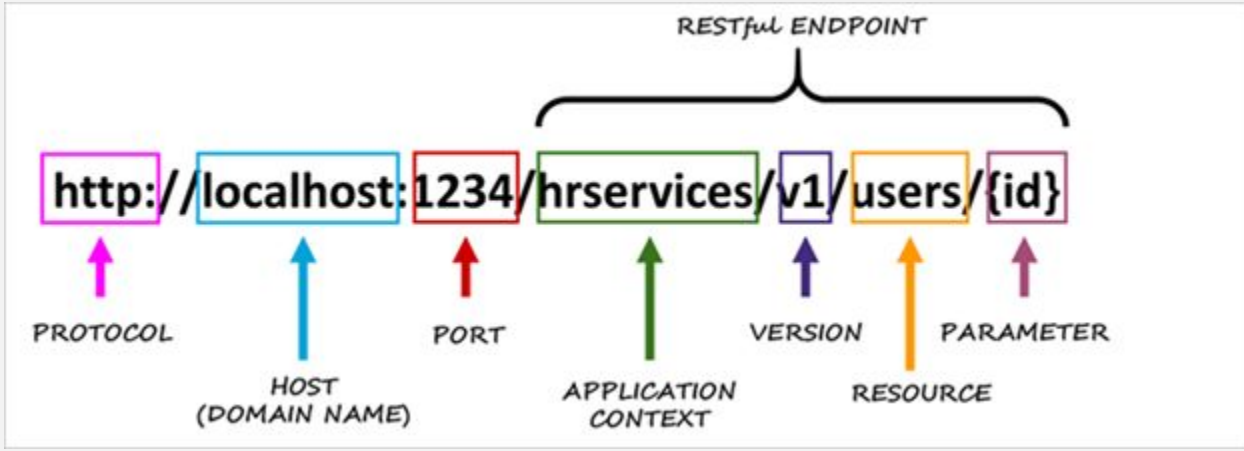
“Most basic RESTful server”

```
const express = require('express')
const app = express()
const port = 3000

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

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

REST URL


RESTful ENDPOINT
http://localhost:1234/hrservices/v1/users/{id}
PROTOCOL
HOST
(DOMAIN NAME)
PORT
APPLICATION
CONTEXT
VERSION
RESOURCE
PARAMETER

Fetch API

“Promises”

```
// Specify the API endpoint for user data
const apiUrl = 'https://api.example.com/users/123';

// Make a GET request using the Fetch API
fetch(apiUrl)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(userData => {
    // Process the retrieved user data
    console.log('User Data:', userData);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

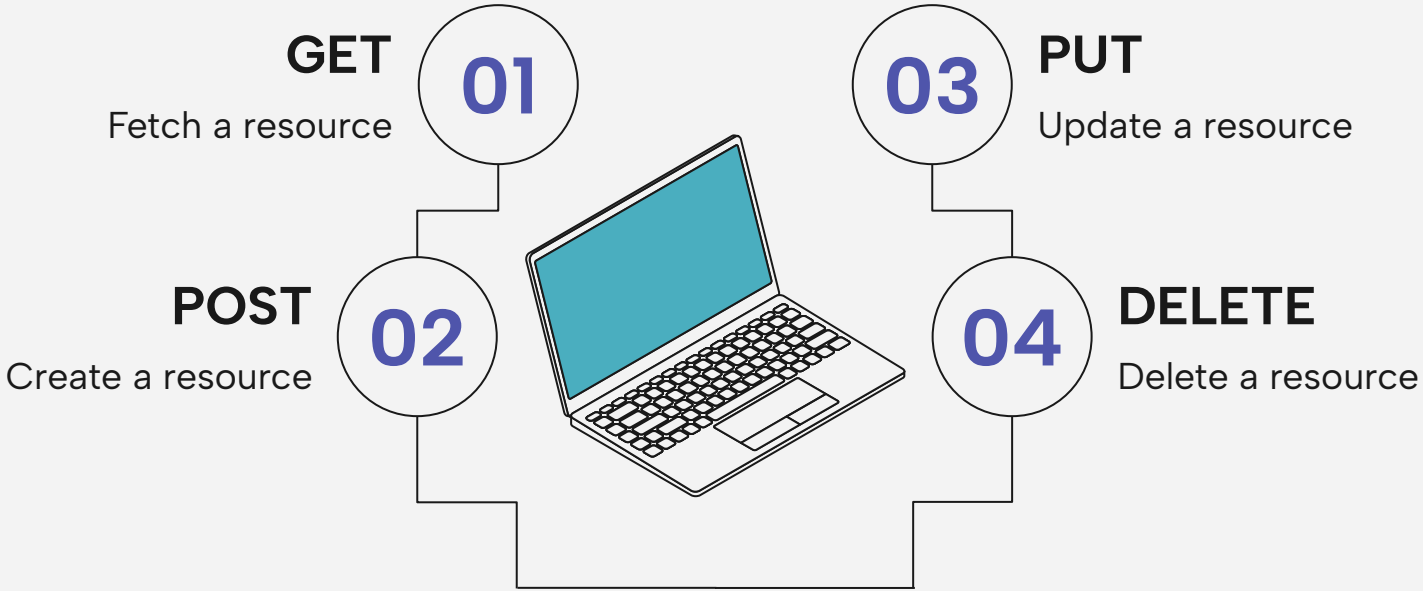
“Async/Await”

```
// Specify the API endpoint for user data
const apiUrl = 'https://api.example.com/users/123';

try {
    // Make a GET request using the Fetch API
    const response = await fetch(apiUrl)
    const userData = await response.json()

    // Process the retrieved user data
    console.log('User Data:', userData)
} catch (error) {
    console.error('Error:', error);
}
```

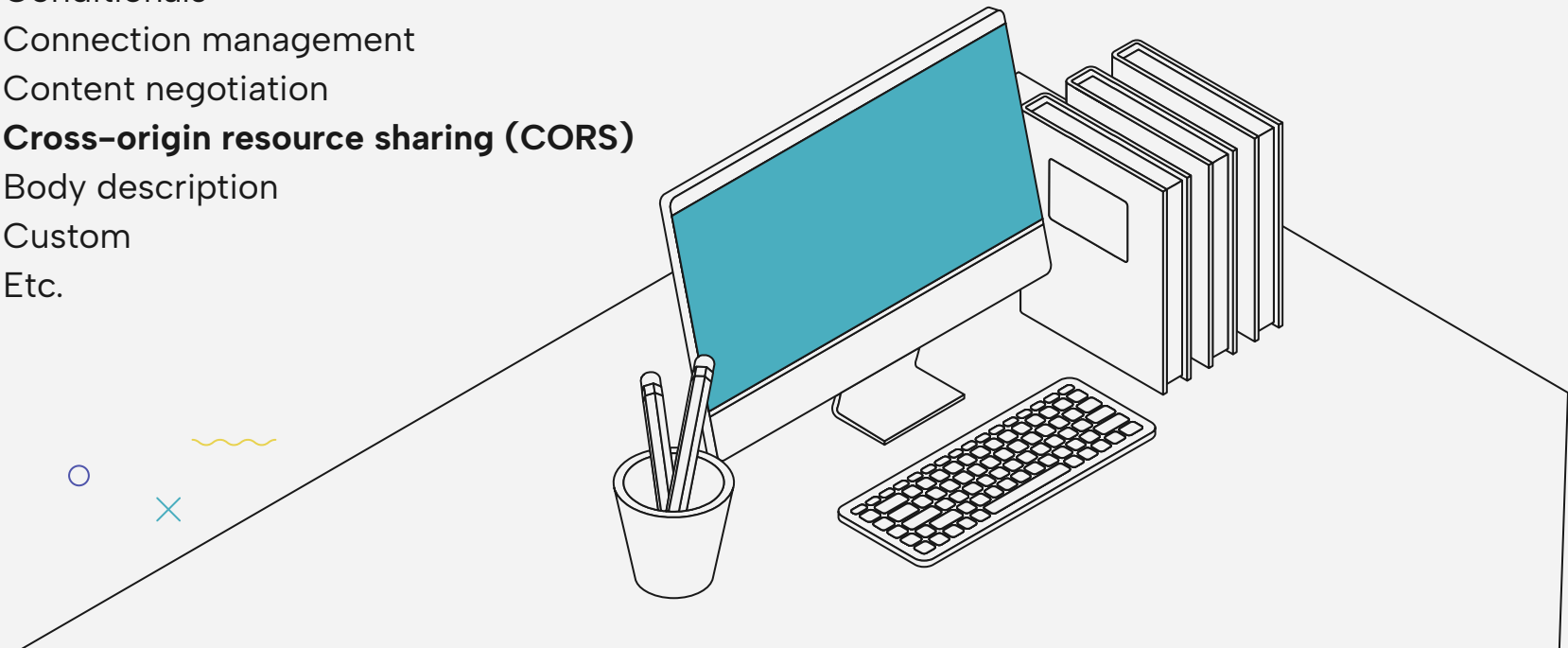
HTTP Methods



"CRUD": Create, Read, Update, Delete

HTTP Headers

- **Authentication**
- Caching
- Conditionals
- Connection management
- Content negotiation
- **Cross-origin resource sharing (CORS)**
- Body description
- Custom
- Etc.



Authentication

Register:

- Validate inputs
- Verify database constraints
- Create a password hash
- Save the new user in the database
- Create a session token
- Store the session token in cookies

Login:

- Validate inputs
- Find the user in the database
- Verify password hashes
- Create a session token
- Store the session token in cookies

Authorization

Using Express.js Middleware...

```
function auth(req, res, next) {
    if (req.cookies.sessionId == undefined) {
        res.status(403) // Forbidden
        return res.send("Forbidden access")
    }

    const user = sessions.get(req.cookies.sessionId);
    if (user == undefined) {
        res.status(403) // Forbidden
        return res.send("Forbidden access")
    }
    req.username = user;
    next();
}
```

Thanks!

Do you have any questions?