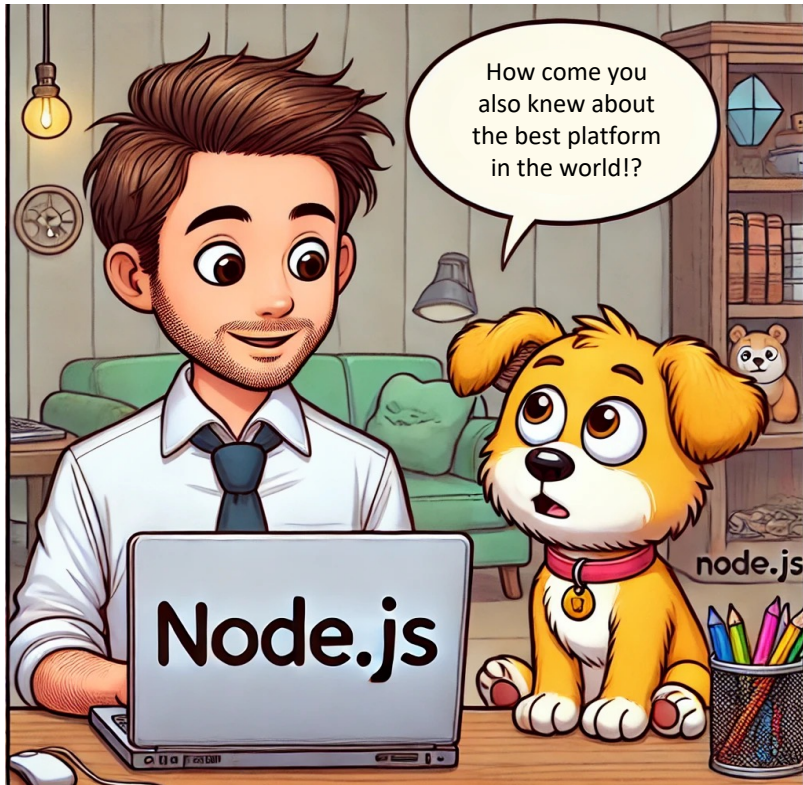




Node.js, Next.js, and APIs

CSC309

Kian Abbasi

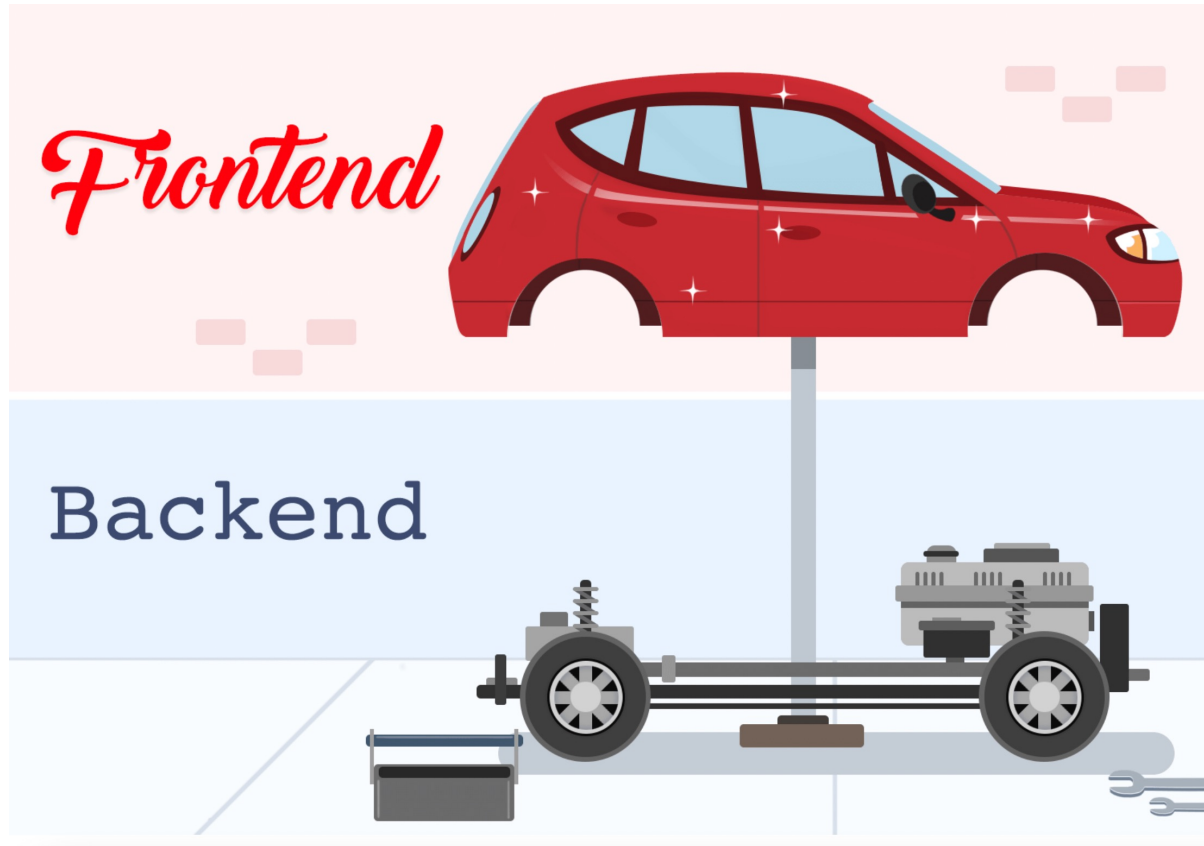
So far

- How **web** works
 - Client vs server
 - HTTP
- **Static** web pages
 - HTML and CSS
- **JavaScript**
 - Syntax, objects, scope, arrow functions
 - **Dynamic** web pages

This session

- Modern architecture of web apps
 - Frontend & backend
 - APIs
- Server-side JavaScript
 - JS projects with Node.js
- Intro to Next.js

Web development

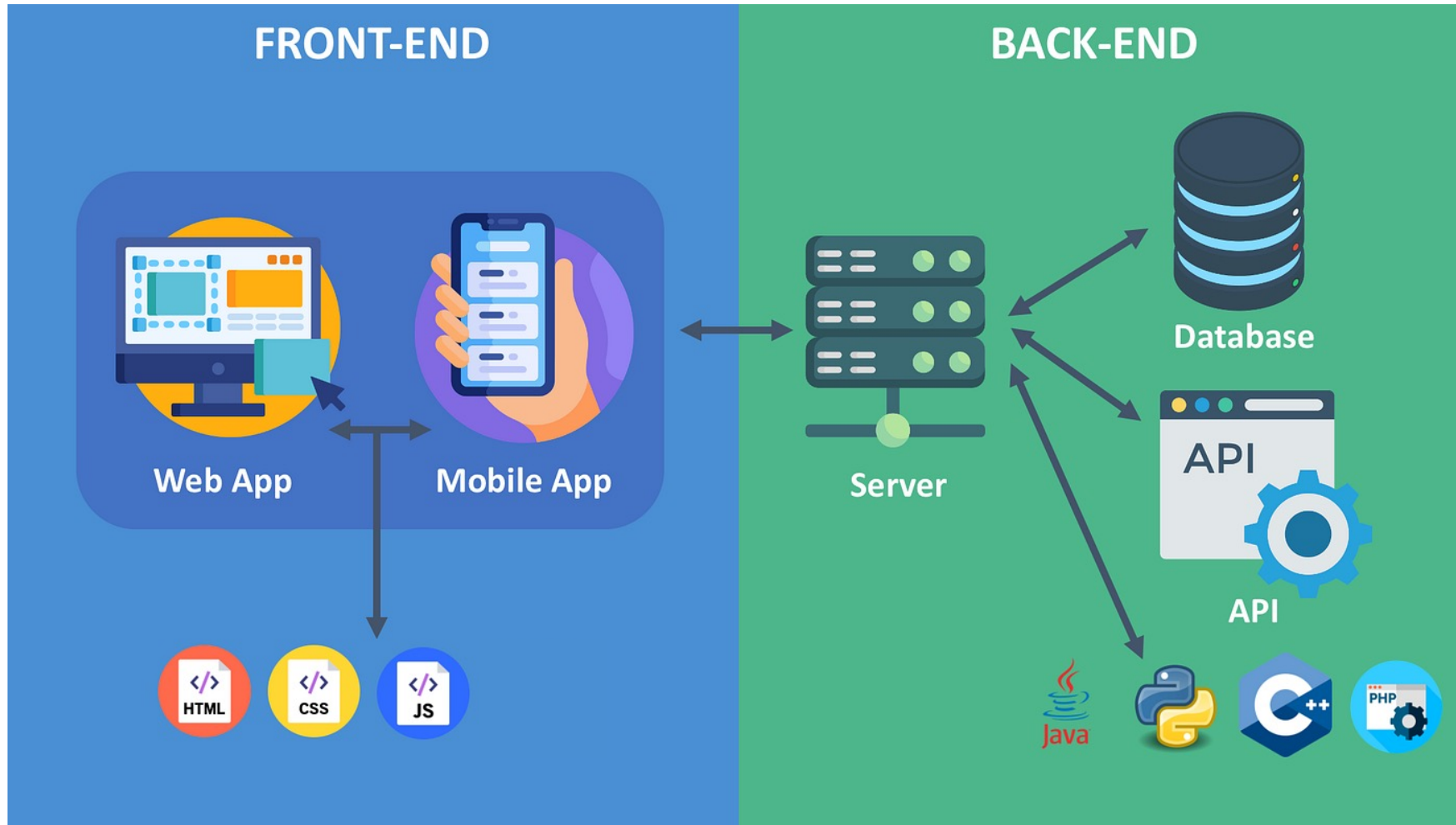


Source: blog.back4app.com



Source: https://www.reddit.com/r/ProgrammerHumor/comments/m187c4/backend_vs_frontend/

Web development



Source: <https://mobilelive.medium.com/backend-for-frontend-basics-a-comprehensive-guide-37768062e55a>

Front-end development

- What user can **see**
 - User interface (**UI**)
 - User experience (**UX**)
- What is run on the **client-side**
 - HTML/CSS rendering
 - **Javascript** codes running on browser

Back-end development

- What user **can't** see
 - What does it even mean?
- All logic and processes that happen **behind the scene**
- Including processing the requests, creating responses, data management, ...
- At the **server-side**!

Web server

- Listens on specified port(s)
- Handles incoming connections
 - Generates a response
 - Fetches a file
 - Forwards them to corresponding applications
- Load balancing, security, file serving, etc
- Examples: Apache, Nginx

Backend frameworks

- Doing everything from scratch?
 - Listen on a port, process http requests (path, method, headers, body), retrieve data from storage, process data, create the response
- Not really a good idea!
- A lot of **frameworks** are out there!
 - A lot of things are pre-implemented

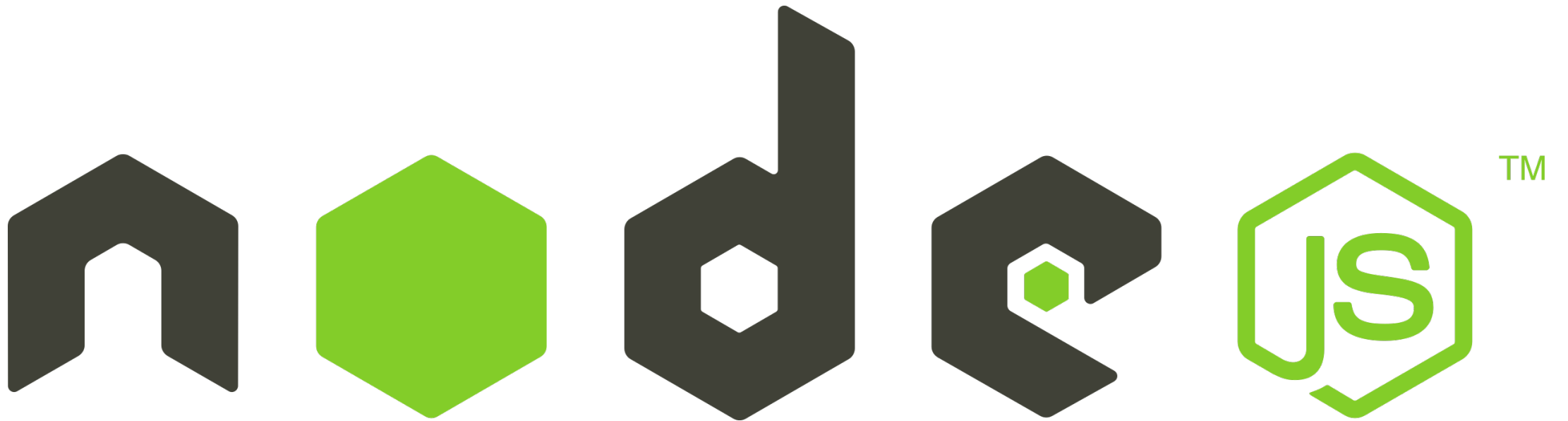
Backend frameworks

- **PHP**: Laravel, CodeIgniter
- **Python**: Django, Flask, FastAPI
- **JavaScript**: Express, Next.js
- **Ruby**: Ruby on Rails

Concept is more important than
framework!

Unique power of JavaScript

- JavaScript is the **only** language that can be used in **both** front-end and back-end codes!
- Helpful for code **consistency**, unity, type sharing, library sharing, etc.
- Thanks to **Node.js**, we can have JavaScript **projects**
 - Could be backend, frontend, or even **both**!



Node.js

- JS does **not** have to be run on the **browser**!
- **Node.js**: a runtime **environment** to for running JS **server-side**
- Includes a **package manager**, **console**, build tools, etc.

Node console

- Opens with the `node` command
- You can execute `inline` JS code
- No `window` or `DOM` object
 - We are outside of the browser
- Files can be run as well
`node <filename>`

Installing libraries

- Node Package Manager (**npm**)
 - Similar to **pip**, **maven**, etc. in other languages
- Install packages via `npm install <package_name>`
 - Packages are **stored** in the **node_modules** directory
- **Automatically** generates and **maintains** a file named **package.json**

Import and export

- Variables, classes, or functions can be **exported** from a JS file (aka **module**)

```
const var1 = 3, var2 = (x) => x + 1  
export { var1, var2 }
```

- Can be **reduced** to **one** statement:

```
export const var1 = 3, var2 = (x) => x + 1
```

- Other modules can **import** them

```
import { var1 } from './App'
```

Default export

- Each module can have **one** **default** export

```
export default App
```

- **Importing** the default export:

```
import App from "./App"
```

- This time, the names do **not** have to **match**
 - Can be imported under any **arbitrary** name

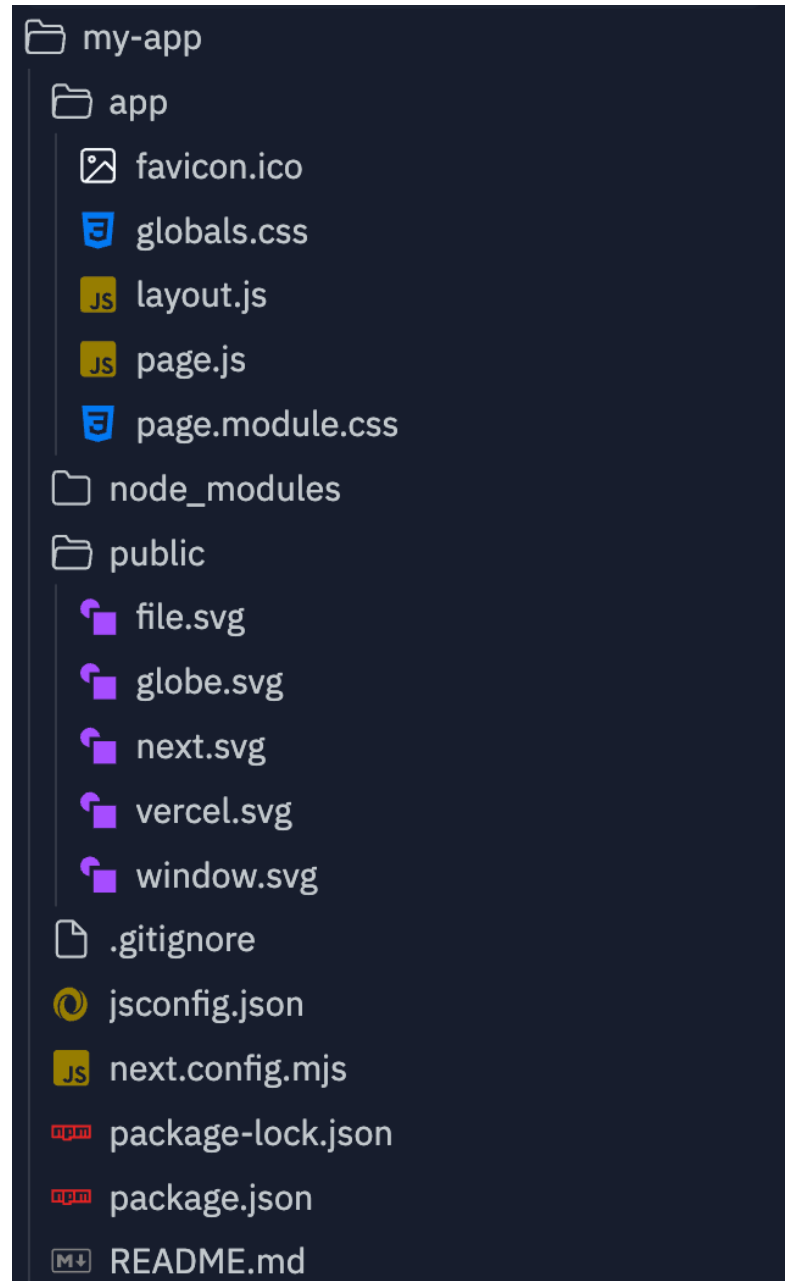
Backend project with Node.js

- In this course, we use **Next.js** as the backend (and frontend) framework
 - More on frontend later in the course!
- Create a project via Node Package Execute (**npx**)
 - Run `npx create-next-app@latest`
 - Answer **Yes** to the prompt about using **App router**
 - You can answer no to other prompts for simplicity!

Next.js project

Visit <https://nextjs.org/docs>

- Looking at **package.json**
 - **next** and **react** are installed
 - Ignore **react** for now
- Create a folder named **api** inside the **app** directory
 - Ignore **everything** except that folder
 - This is our main **backend** directory!



API directory

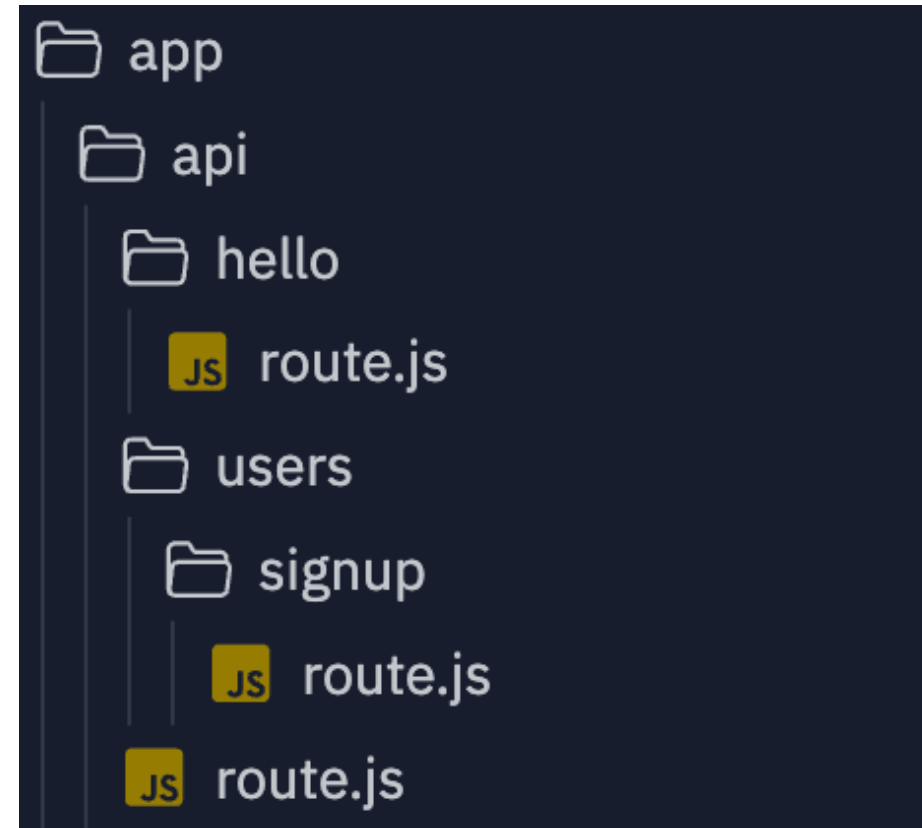
The concept of API is discussed later today!

- In the **app** directory, every **directory** will correspond to a **URL path**, if it contains a **route.js** file

/api/users/signup

/api/hello

/api



API handler (aka URL handler)

- Define an **export function** for each accepted **HTTP method**
 - Executed when a request with the corresponding **URL path** and method arrives
- Inputs the **request** and returns a **response**

- Example

```
export async function GET(request) {  
  // Backend logic  
}
```

Status & Responses

- Text response

```
new Response("Hello world")
```

- HTML response

```
new Response(`  
  <html>  
    <body>  
      <h1>Login Successful!</h1>  
    </body>  
  </html>  
`,  
  { headers: { "content-type": "text/html" } }  
);
```

Status & Responses

- Error response

```
new Response("Bad request", { status: 400 });  
new Response("Server error", { status: 500 });
```

- Redirects

```
redirect('/api/users/login')
```

- JSON response

```
NextResponse.json({ message: "Hello, world!" });
```


Request object

- Headers

```
if (request.headers.get('content-type') === 'application/json')
```

- Query params (after ? in the URL)

```
const search = request.nextUrl.searchParams.get('search')
```

- Request body (if in JSON format)

```
const { username, password } = await request.json()
```

URL parameters

- **Dynamic segments** in the URL
 - e.g., `/stores/1/products`, `/stores/2/products`
- Defined in **directory names**
 - `/app/api/stores/[storeId]/products/route.js`
- Can be access via the **params** object

```
export async function GET(request, { params }) {  
  const { storeId } = await params;
```

How a web app works

- Browser **requests** a URL
- API handler returns a **giant HTML** with the appropriate CSS, and client-side JS content
 - Potentially reads it from a **separate** file
 - Could be dynamically filled based on the request
- Users clicks on **links** and/or fills out **forms**
- API handlers **process** the GET/POST requests and returns **another** HTML response or a redirect

Does it, really?

Web apps worked this way before 2010s!

Some still do...

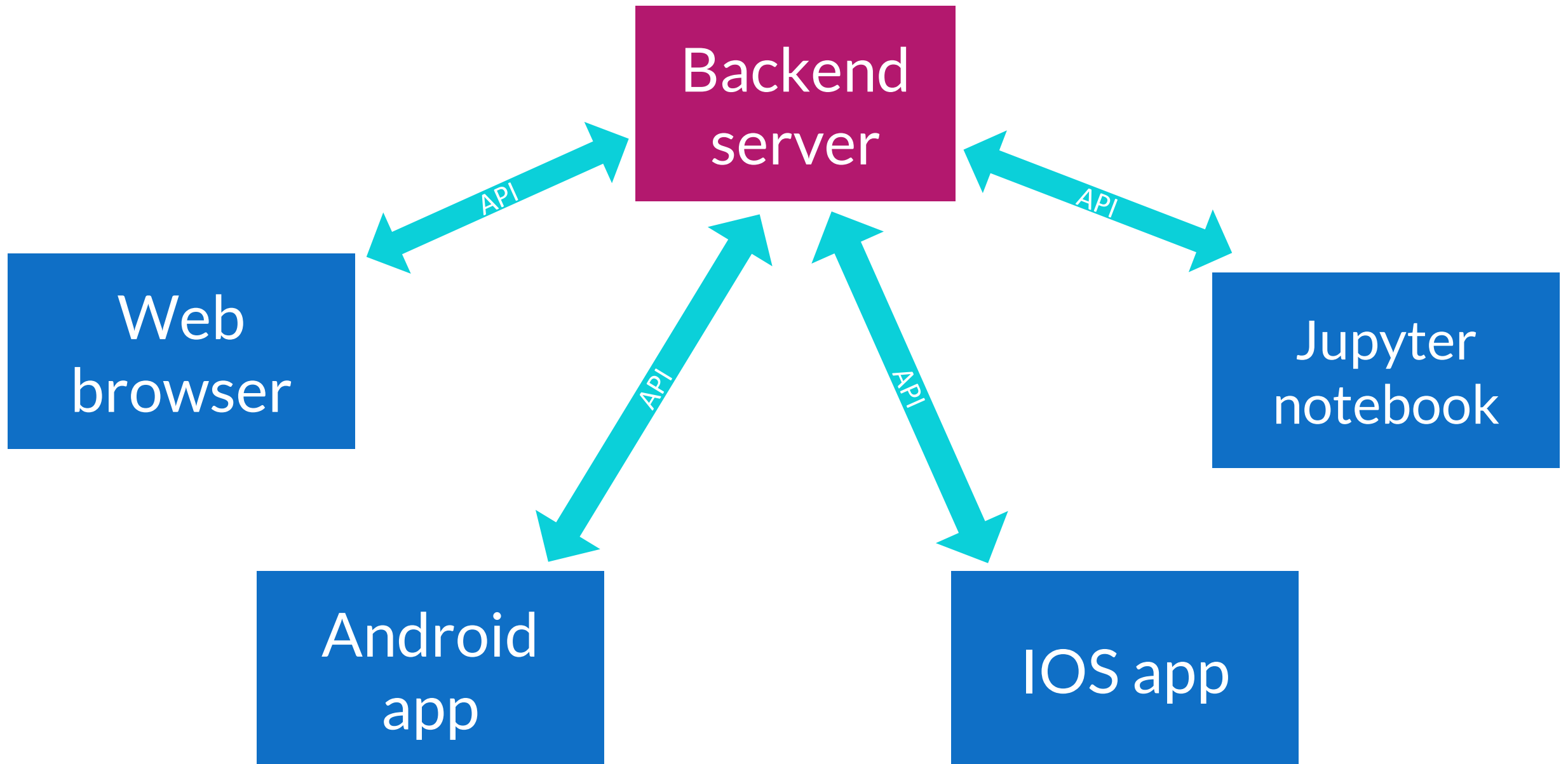
Can you think of some major drawbacks of this style of web apps?

Drawbacks

- Too **backend-oriented**
 - All frontend logic is served as part of backend **handlers**
 - Code gets messy and hard to understand
- **Limits** all frontends to web browsers
 - What about mobile, watch, assistant, etc. ?
- Frontend can't be as **sophisticated**
 - Example: Single-page application

Separate backend and frontend

- **Modularity**
 - Changes in frontend will **not** affect backend and vice versa
- **Consolidation**
 - One backend and **multiple** frontends (web, android, iOS)



Modularity

- Different services/apps **talk** to each other with a **protocol**
- **API**: The way an application can be talked to
 - Stands for **Application Programming Interface**
- **Web** applications: typically, a set of **HTTP** requests

Separate Backend and Frontend

- Backend views are only about data **retrieval** and **manipulation**
- Backend does **not** care about how data is **shown**, UI, or UX
- No HTML, CSS, client-side JS

Response format

- A popular standard is JavaScript Object Notation or JSON
 - Derived from JavaScript syntax for defining objects
- Easy, human-readable, and fast
 - Many languages (python, javascript, ...) have built-in parsers and support

JSON

- **Primitive** types: number, string, boolean, null
- Array: **ordered** collection of elements
- Object: **key-value** pairs
 - Keys are **strings**
- Array **elements** and object **values** can be of **any** type (string, null, array, object, etc.)

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Source: wikipedia

API architecture

- Representational State Transfer (REST)
- A set of URL endpoints that do a data management task
 - Login, signup, list of comments, create a post, edit profile, ...
- All data is in the JSON format
 - Both request payload and response
- Example:
exchange-docs.crypto.com/exchange/v1/rest-ws/index.html

Restful APIs in Next.js

- Next.js **natively** supports Restful APIs
- Request body is **parsed** from JSON into a **JS object**
 - Accessible via `await request.json()`
- Native support for returning JSON **response**
 - Via `NextResponse.json(...)`
 - Appropriate **headers** are also set

Next session

- Data management
 - **Model** design
 - The MVC design pattern
- **ORMs**
- Querying the database in Next.js **API handlers**
 - **CRUD**