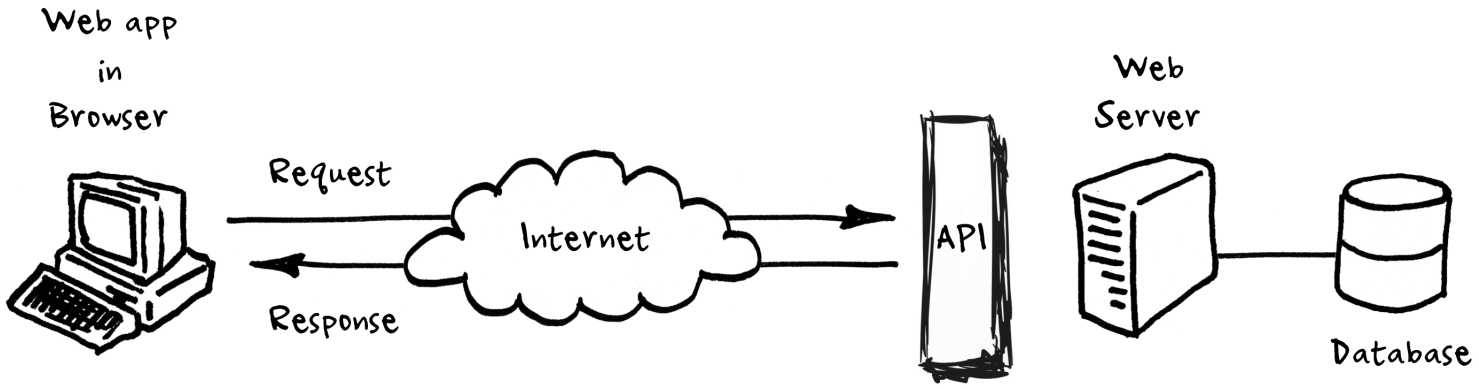
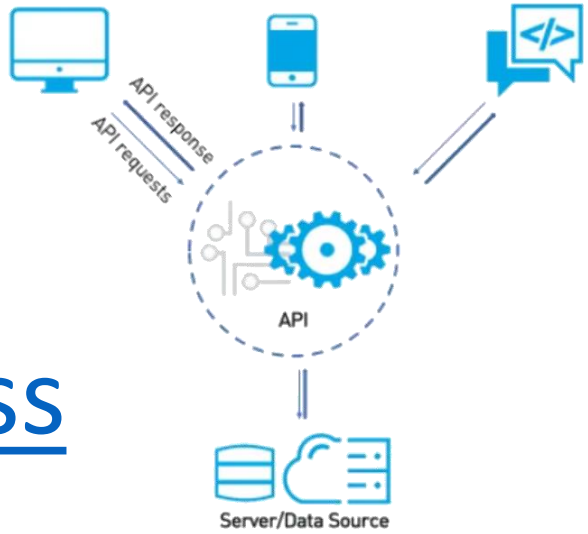
Web API using JavaScript



Outline



1. Web API using express
2. Web API using Next.js
3. HTTP Data Exchange Mechanisms

Course Roadmap

Web Client

Request

Response

Web Server

Frontend development

HTML for page content & structure

CSS for styling

JavaScript for interaction

Backend development

We are HERE

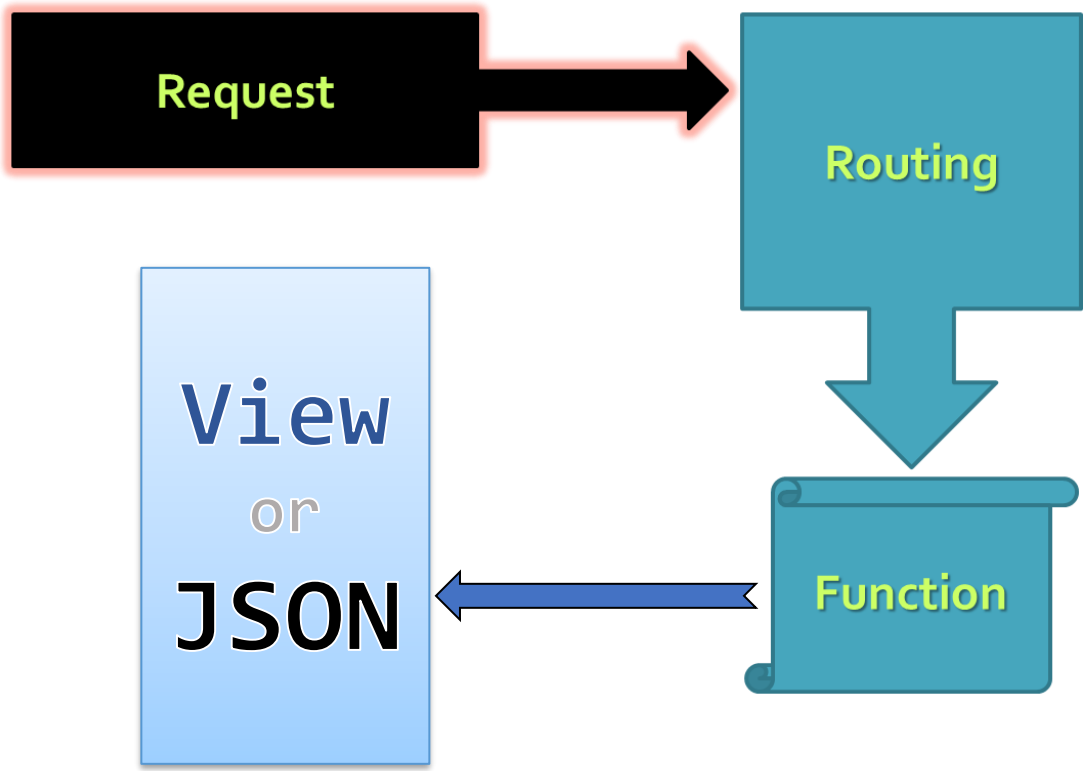
Web API

Web Pages

Data Management

Prisma

Web API using Node.js Express



What is a Web API

- Web API: A set of methods exposed over the web via HTTP to allow **programmatic access to applications**
- Web API are designed for **broad reach**:
  - Can be accessed by a broad range of clients including browsers and mobile devices
  - Can be implemented or consumed in any language
- Uses HTTP as an **application protocol**

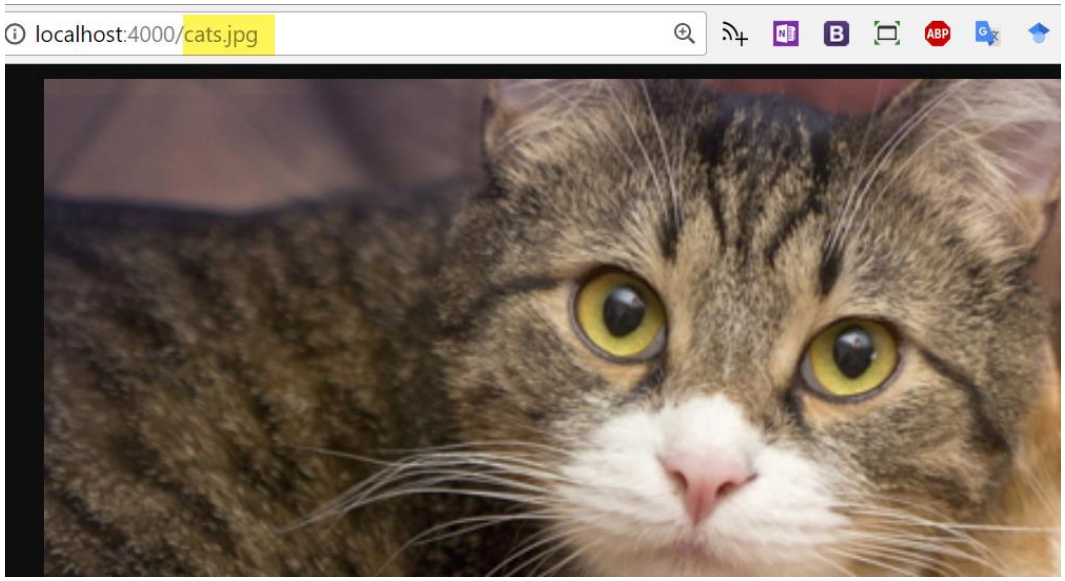
Create and Start an Express App

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

app.get('/', (req, res) => {
    res.send('السلام عليكم ورحمة الله وبركاته’);
});

const port = 3000;
app.listen(port, () => {
    console.log(`App is available @ http://localhost:${port}`)
});
```

- The app listens for incoming request @ http://localhost:3000/
- When someone visits this Url the function associated with **get** **'/'** will run and **'السلام عليكم ورحمة الله وبركاته'** will be returned to the requester

Serving Static Resources using Express

- To serve up static files, the **express.static** middleware function is used with the folder path of the files to be served



Routing

- Requests can be routed based on:
  - **HTTP Verb** – GET, POST, PUT, DELETE
  - **URL Path** – e.g., /users
- App Route **maps** an **HTTP Verb** (e.g., GET or POST) + a **URI Path** (like /users/123) to a **route handler function**
  - The handler function is passed a **req** and a **res** objects
  - The **req** object represents the HTTP request and has the query string, parameters, body and HTTP headers
  - The **res** object represents the HTTP response and it is used to send the generated response

Path Parameters

- Named **path parameters** can be added to the URL path. E.g., **/students/:id**
- **req.params** is an object containing properties mapped to the named path parameters
  - E.g., if you have the path **/students/:id**, then the "id" property is available as **req.params.id**

```
app.get('/api/students/:id', (req, res) => {
    const studentId = req.params.id;
    console.log('req.params.id', studentId)
})
```

```
app.get('/authors/:authorId/books/:bookId', (req, res) => {
    // If the Request URL was http://localhost:3000/authors/34/books/8989
    // Then req.params: { authorId: "34", bookId: "8989" }
    res.send(req.params);
})
```

Query Parameters

- Named **query parameters** can be added to the URL path after a ? E.g., **/posts ?sortBy=createdOnDate**
- Query parameters are often used for **optional** parameters (e.g., optionally specifying the property to be used to sort of results)
- **req.query** is an object containing a property for each query parameter in the URL path
  - If you have the path /posts**?sortBy**=createdOnDate, then the "**sortBy**" property is available as req.query.**sortBy**

```
app.get('/api/students?sortBy=studentId', (req, res) => {
    // req.query.sortBy  => "studentId"
    const sortBy = req.query.sortBy
    console.log(req.query.sortBy', sortBy)
})
```


10

Working with a Request Body

- To access the request body a middleware is used to parse the request body
- **express.json()** is a middleware function that extracts the body portion of an incoming request and assigns it to **req.body**

```
import express from 'express';
const app = express();
app.use( express.json() );
app.post('/heroes', async (req, res) => {
    const hero = req.body;
    await heroRespository.addHero(hero);
    res.status(201);
});
```

Express Router

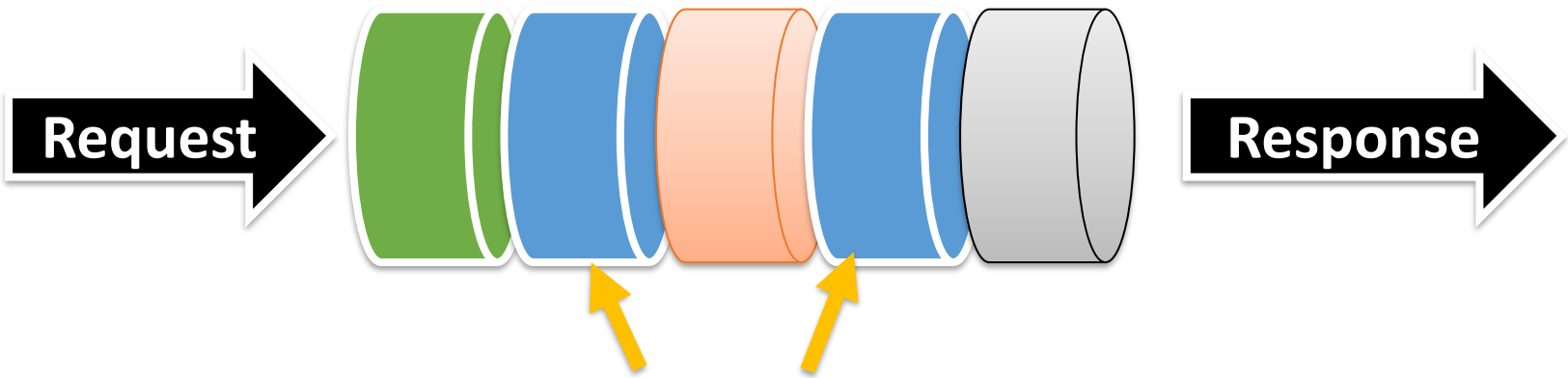
- For simple app routes can defined in **app.js**
- For large application, Express Router allows defining the routes in a separate file(s) then attaching routes to the app to:
  - Keep *app.js* clean, simple and organized
  - Easily find and maintain routes

```
// routes.js file
const router = express.Router()
router.get('/api/students', studentController.getStudents )
module.exports = router

//app.js file - mount the routes to the app
import { router } from './routes.js';
app.use('/', router);
```

Express Middleware

- Express middleware allows **pipelining** a request through a series of functions.
  - Each middleware function may **modify** the request or the response
- Request Processing Pipeline: the request passes through an *array* of functions before it reaches the route handler

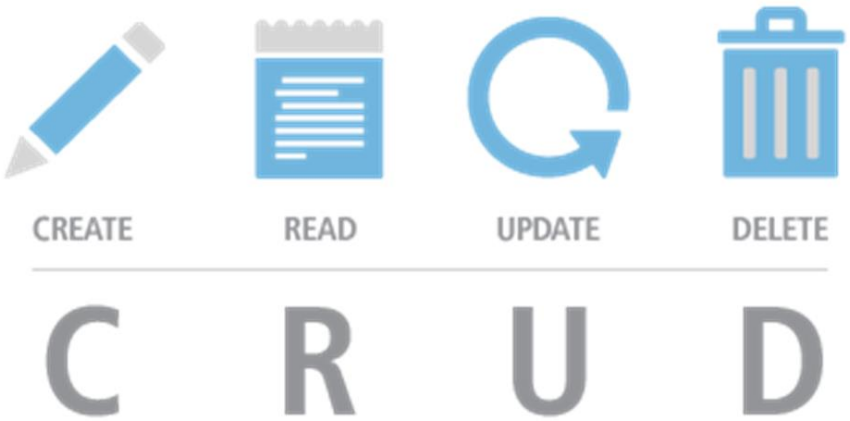
```
/* express.json() is a middleware function that extracts the body portion of
an incoming request and assigns it to req.body.
 */
app.use( express.json() );
```



Middleware (body parser, logging, authentication, router etc.)

Custom Middleware Example

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

//Define a middleware function
function logger (req, res, next) {
    req.requestTime = new Date();
    console.log(`Request received at ${req.requestTime}`);
    next();
}

// Attach it to the app
app.use(logger);

app.get('/', function (req, res) {
    const responseText = `Hello World! Requested at: ${req.requestTime}`;
    res.send(responseText);
})
```



Implementing CRUD Operations

CRUD Operations

- See the posted Hero and Student Examples

```
import heroService from './services/HeroService.js';

//Heroes Web API
router.route('/heroes')
    .get( heroService.getHeroes )
    .post( heroService.addHero );

router.route('/heroes/:id')
    .get( heroService.getHero )
    .put( heroService.updateHero )
    .delete( heroService.deleteHero );
```

Summary

- Express is a popular and easy to use web framework
- It makes building an Http Server and Web API a lot easier
- Provides routing and static content delivery out of the box
- Uses **express.json()** middleware to parse the request body

Resources

- Express Documentation

https://expressjs.com/en/5x/api.html

- Web API Design

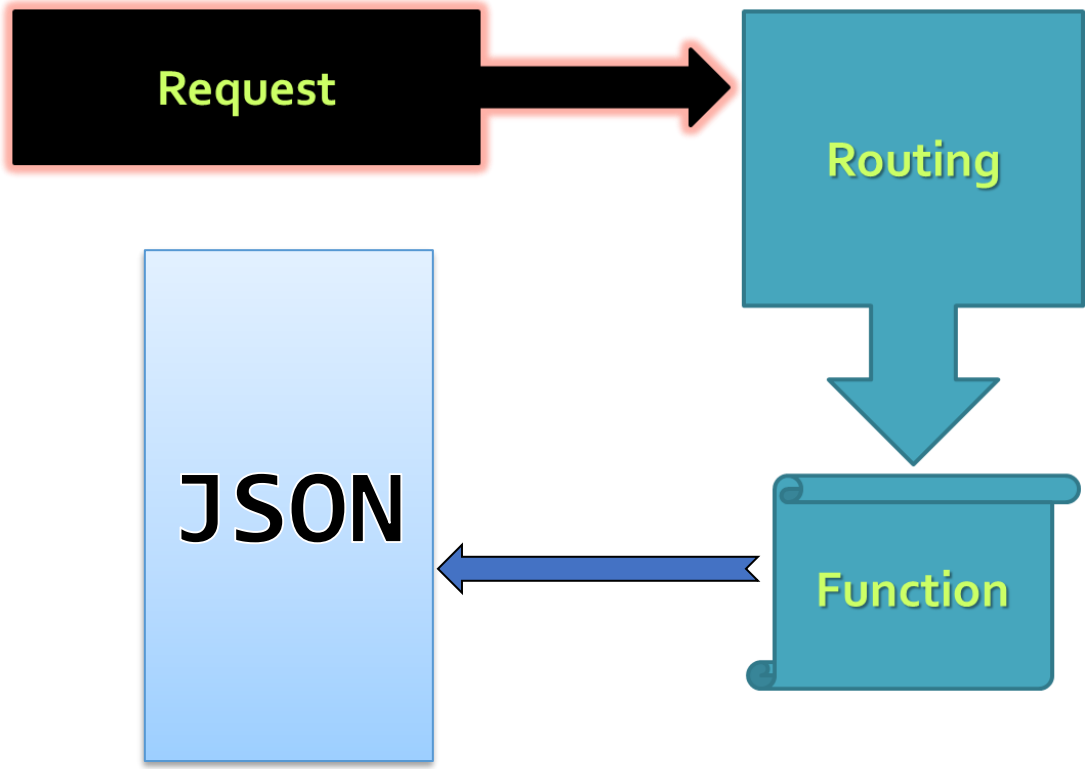
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

- https://cloud.google.com/files/apigee/apigee-web-api-design-the-missing-link-ebook.pdf

- Mozilla Developer Network

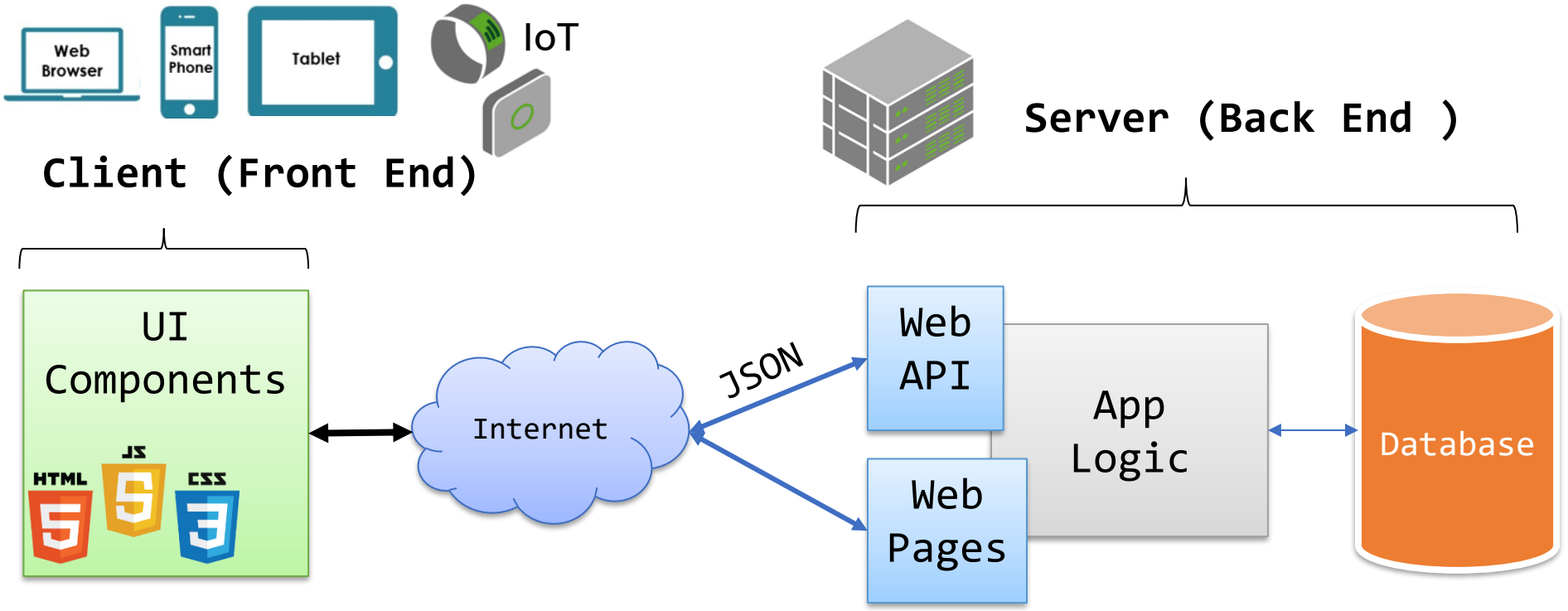
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs

Web API using NEXT.JS



Web App Architecture

- Front-end made-up of **multiple UI components loaded** in response to user actions
- Back-end Web API and Web pages



Next.js Introduction

- React is a JavaScript library for building user interfaces by composing UI components
  - Released in 2013 by Facebook, current version: 19
- Next.js is a React-based framework for building Web API and Web pages
  - Released in 2016 by Vercel, current version: 15
  - Next.js is extends React with features like **server-side rendering** (SSR), and **file-based routing**
  - React is a library for building UI components, whereas Next.js is a full-stack framework for building complete web applications

Next.js - Getting started

- Create an empty folder (with no space in the name use **dash -** instead)
- Create next.js app (select **No** for all questions)

```
npx create-next-app@latest .
```

```
√ Would you like to use TypeScript with this project? ... No / Yes
√ Would you like to use ESLint with this project? ... No / Yes
√ Would you like to use `src/` directory with this project? ... No / Yes
√ What import alias would you like configured? ... @/*
```

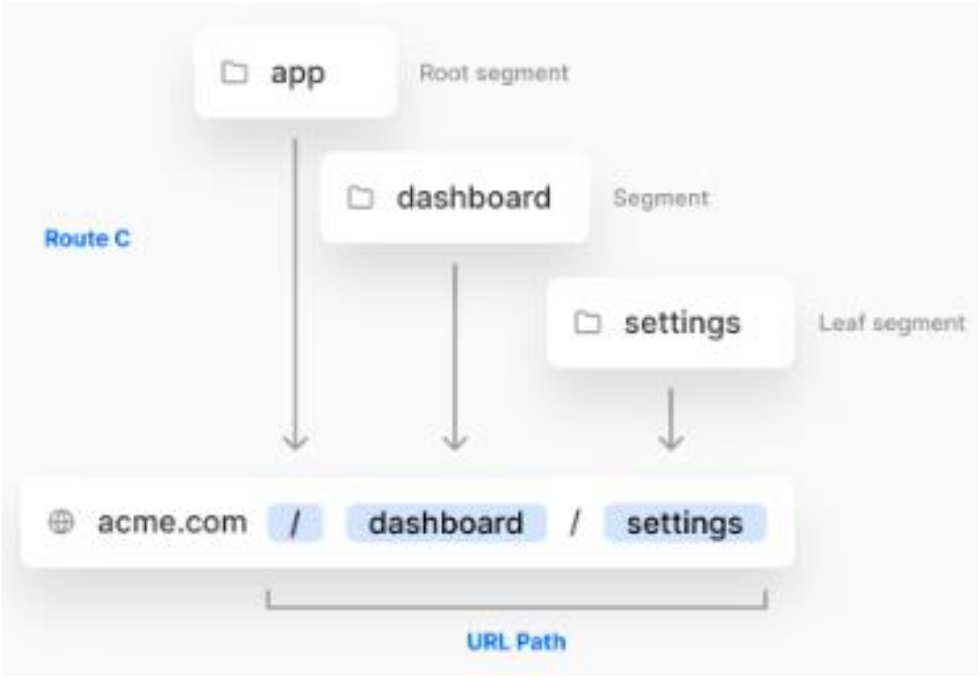
- Creates a new **`Next.js`** project and downloads all the required packages
- Run the app in dev mode: **`npm run dev`**

Next.js Routing

- Next.js has a **file-system based App Router**:
  - Folders inside the **app** directory are used to define routes
    - A route is a single path of nested folders, from the root folder down to a leaf folder
  - Files are used to create Web pages (`page.js`) or Web API (`route.js`)

- Each folder in the subtree represents a route segment in a URL path

- E.g., create `/dashboard/settings` route by nesting two subfolders in the app directory



API Routes

- Add a **`route.js`** file under the **app** folder or within subfolders to define API routes
- A route.js file can export async functions named after HTTP methods (`GET`, `HEAD`, `OPTIONS`, `POST`, `PUT`, `DELETE`, `PATCH`) to handle requests
- Any subfolder within app containing a route.js file is treated as a Web API endpoint (e.g., app/api/hello/route.js).

```
export async function GET(request) {
  return new Response('Hello, Next.js!');
}
```

- Visiting http://localhost:3000/api/hello will return Hello, Next.js!

Routing in Next.js

- Requests can be routed based on:
  - **HTTP Verbs**: GET, POST, PUT, DELETE
  - **URL Paths**: e.g., /users
- An App Route maps an HTTP Verb (e.g., GET, POST) and a URL path (e.g., /users/123) to a **route handler function**
  - The handler function receives a **request** object and returns a **response** object
  - The **request** object allows extracting data, such as the request body
  - The **response** object represents the HTTP response and is used to send the generated output

25

HTTP Data Exchange Mechanisms

Summary of HTTP Data Exchange Mechanisms

Method	Description	Use Case	Example
Path Params	Dynamic segments in the URL path.	Identifying specific resources.	/users/{id} → /users/123
Query Params	Key-value pairs in the URL after ?.	Filtering, sorting, and pagination.	/products?category=laptop&sort=price
Request Body	Contains structured data (JSON, XML, form data).	Sending data in POST, PUT, PATCH requests.	{ "name": "John Doe", "email": "john@example.com" }
Cookies	Small data stored in the browser, auto-sent with requests.	Authentication, session management, preferences.	Set-Cookie: sessionId=abc123; HttpOnly

Path parameters using Dynamic Routes

- Dynamic Routes: Wrap folder names in square brackets **[folderName]** to define **path parameters**

Route	Example URL	params
app/blogs/[id]/route.js	/blogs/123	{ id: '123' }
	/blogs/234	{ id: '234' }

- Path parameters are part of the URL structure to **identify** a resource
  - Used when the value is required & unique
- Path Parameters are passed as **params** in handler functions

```
// app/blogs/[id]/route.js
export async function GET(request, { params }) {
  const id = (await params).id;
  return new Response(`Blog id# ${id}`)
}
```

Catch-all Segments

- **Catch-All Segments**: Use an ellipsis inside the brackets `[...folderName]` to match all sub-paths under a route
  - Allow flexible path matching
- `params` returns an array of matched segments
  - e.g., catch-all segments in **app/blogs/[...filter]** will match any path underneath **/blogs** such as: /blogs/2025, /blogs/2025/3/10

Route	Example URL	params
app/blogs/**[...filter]**/route.js	/blogs/2025	{ filter: ['2025'] }
	/blogs/2025/3	{ filter: ['2025', '3'] }
	/blogs/2025/3/10	{ filter: ['2025', '3', '10'] }
app/shop/**[...slug]**/route.js	/shop/clothes	{ slug: ['clothes'] }
	/shop/clothes/tops	{ slug: ['clothes', 'tops'] }
	/shop/clothes/tops/shirts	{ slug: ['clothes', 'tops', 'shirts'] }

Optional Catch-all Segments

- Catch-all Segments can be made optional by including the parameter in double square brackets:
  `[[...folderName]]`
  - For example, **`app/shop/[[...slug]]/route.js`** will also match **`/shop`**, in addition to `/shop/clothes`, `/shop/clothes/tops`, `/shop/clothes/tops/shirts`
  - The difference between catch-all and optional catch-all segments is that with optional, the route without the parameter is also matched (`/shop` in the example above)

Route	Example URL	params
app/shop/**[...slug]**/route.js	/shop	{ slug: undefined }
	/shop/clothes	{ slug: ['clothes'] }
	/shop/clothes/tops	{ slug: ['clothes', 'tops'] }
	/shop/clothes/tops/shirts	{ slug: ['clothes', 'tops', 'shirts'] }

URL Query Parameters

- Query Parameters: Key-value pairs in the URL after **?**
- Used for **filtering**, **sorting**, and **pagination**. Example: /products?**query=laptop&sort=price**
- Optional and flexible: Modify requests without changing the URL structure
- Accessed in Next.js using `request.nextUrl.searchParams` returns a map of query parameters
  - For /products?**sort=price**, access the sort parameter using:

```
const sort = request.nextUrl.searchParams.get("sort");
```


31

Working with a Request Body

- Request Body: Contains the main data payload in POST, PUT, and PATCH requests
- Support multiple formats such as JSON, XML, and form data
  - E.g. { "name": "John Doe", "email": "john@example.com" }
- Use request .json(), .text(), or .formData() to retrieve the request body

```
export async function POST(request) {
    let newHero = await request.json();
    newHero = await addHero(newHero);
    return new Response( ...,  {status: 201 }) ;
}
```

Request Body FormData

- You can read the FormData using the `request.formData()` function

```
export async function POST(request) {
  const formData = await request.formData();
  const email = formData.get("email");
  const password = formData.get("password");
  const user = verifyUser(email, password);
  if (user) {
    return new Response(JSON.stringify(user), { status: 200 });
  } else {
    return new Response(JSON.stringify({ message: "Invalid
            credentials" }), { status: 401 });
  }
}
```

Headers

- Headers: Metadata in requests & responses
  - Used for authentication, security, content negotiation
  - Example: `apiKey: ab13579xyz`
- You can read http headers with the **`headers`** library from `next/headers` package
  - You can also return a new Response with headers

```
import { headers } from "next/headers";
export async function GET(request) {
  const headersList = await headers();
  const apiKey = headersList.get("apiKey");
  return new Response("Hello, Next.js!", {
    status: 200,
    headers: { apiKey: apiKey || "No API Key" },
  });
}
```

Cookies

- Cookies are small name-value pairs stored in a user's browser by a website
- They enable websites to remember user preferences, authentication states, shopping cart items, and more.
- Common Uses of Cookies:
  - Session Management – Keep users logged in
  - Personalization – Remember language preferences
  - Tracking & Analytics – Monitor user behavior such as track visits, clicks, and interactions (e.g., Google Analytics)
- Once set, cookies are automatically sent with each request to the originating website

Cookies vs. Headers

- Both cookies and HTTP headers help manage state and data in web applications, but they serve different purposes:

Aspect	Cookies	HTTP Headers
Purpose	Store small user data in the browser (e.g., sessions, preferences)	Pass metadata between client and server (e.g., authentication, caching)
Storage Location	- Stored in the user's browser and sent with every request - Can persist across sessions (based on max-age or expires)	Exists only for a single request/response
Use Cases	Session management, personalization, tracking	Security (authentication), caching, content negotiation

When to Use Each?

✅ **Use Cookies When:**

1.User Authentication: Storing session tokens or login state so users stay logged in across pages

2.User Preferences: Saving dark mode settings, language preferences, etc

3.Shopping Cart Persistence: Keeping cart items saved even after closing the browser

✅ **Use HTTP Headers When:**

1. Security & Authentication: Using **Authorization: Bearer <token>** for API authentication instead of cookies

2. Content Negotiation: Using **Accept: application/json** to tell the server the expected response format

4. Caching: e.g., Cache-Control: max-age=3600 header tells the browser to store the response for 3600 seconds (1 hour) before requesting a fresh copy from the server

Summary - HTTP Data Exchange Mechanisms

Feature	Where Used?	When to Use?	Example
Path Params	URL	Required, unique resource identification	/users/123
Query Params	URL	Optional, filtering, sorting, pagination	/search?query=laptop
Cookies	Browser & HTTP requests	Session, authentication, preferences	session_id=xyz123
HTTP Headers	Request/Response metadata	Authentication, content negotiation	Authorization: Bearer token
Request Body	Request payload	Sending structured data in POST, PUT, PATCH	{ "name": "John" }

Summary

- Next.js = React-based full stack web framework that allows creating server-side rendered pages, and Web API
- Next.js has a **file-system based router**: when a folder is added to the **app** directory, it's automatically available as a route
  - In Next.js you can add brackets to the folder name to create a dynamic route
- Add Web API Route Handlers inside the app/ directory (e.g., app/api/users/route.js)

Summary (continued…)

- Build a public API if it need
  - to be shared by web, mobile, or third-party clients to consume your data/functionality
  - to proxy a backend /external service and include secret authentication headers
- Export HTTP methods (GET, POST, PUT, DELETE, etc.) in the route.js file
- Use Web Standard APIs to interact with the Request object and return a Response
- Fetch the API routes from the client with fetch('/api/...')

Resources

- Learn Next.js

https://nextjs.org/docs

- Next.js Web API

https://nextjs.org/blog/building-apis-with-nextjs

- Next.js blog

https://nextjs.org/blog