React

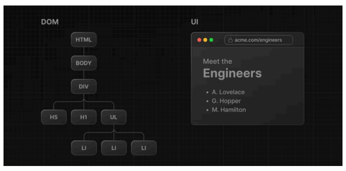
Web App Building Blocks
* User Interface: how users consume/interact w/ app
* Routing: how users navigate parts of app
* Data Fetching: where data lives/how to get it
* Rendering: when/where you render static/dynamic content
* Integrations: 3rd-party services/how to connect to them
* Infrastructure: where you deploy, store, run app code (serverless, CDN, edge(close to client), etc)
* Performance: optimizing app for end-users
* Scalability: how app adapts as team, data, traffic grow
* Developer Experience: team's experience building/maintaining app

React: js library for building interactive user interfaces **(UI)**

APIs: helpful functions

Nextjs: React framework that gives building blocks for web apps

Document Object Model (DOM): object representation of HTML elements, bridge btwn code and UI, tree-like structure w/ parent/child relationships



* HTML represents initial pg content, while DOM represents updated pg content

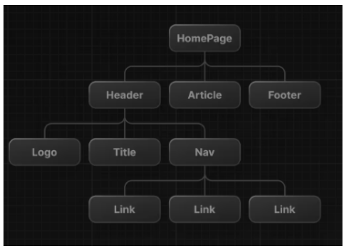
Imperative vs Declarative Programming
* Imperative: giving chef step-by-step how to make pizza
* Declarative: ordering a pizza w/out concerns abt steps making it

React
react: core React library
react-dom: provides DOM-specific methods that enable you to use React w/ DOM

JSX: syntax extension for js allows you to describe UI in familiar HTML-like syntax
* browsers don't understand JSX out of the box -> need js compiler

React Core Concepts
* Components: Lego bricks of UI, are functions in React
* Props: read-only properties that can be passed to react components, **one-way data flow** data flows down component tree
* State: UI info that changes over time, usually triggered by user interaction

Nesting Components
*Component Trees



Using Variables in JSX
* inside curly braces **{}** can add js expression that evaluates to single value ie
1. **Object property w/ dot notation**
{props.title}
2. **Template literal**
{`Cool ${title}`}
3. **Returned function value**
function createTitle(title) {
If (title) {
return title;
} else {
return 'Default title';
}
}

{createTitle(title)}
4. **Ternary Operators**
{title ? title : 'Default title'}

Event Handlers: ie functions to "handle" events when triggered

State and hooks
* State: UI info that changes over time, usually triggered by user interaction
* Hooks: allow you to add additional logic to components

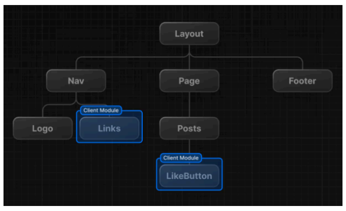
Note: unlike props which are passed to components as first function parameter, state is initiated and stored w/in component

* You can pass state info to children components as props, but logic for updating state should be kept w/in component where state initially created

StackBlitz
https://stackblitz.com/edit/vitejs-vite-9svwuax9?file=index.html

Environments
* Client: browser on user's device that sends request to server for app then turns response from server into UI
* Server: computer that stores app, receives client requests, does computation, sends back response

Network Boundary: separation btwn different environments

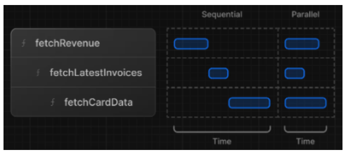


* in react, can choose where separation is
* components split into 2 module graphs **server module graph/tree** and **client module graph/tree**
* after server components rendered, special data format called **React Server Component Payload (RSC)** sent to client

React Server Component Payload (RSC): contains:
1. Rendered server components
2. Placeholders for where client components should be rendered and refs to their js files

* Nextjs uses server components by default

Request waterfall: sequence of network requests that depend on completion of previous requests

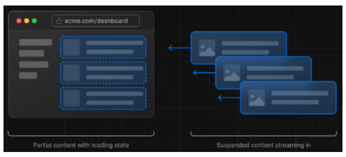


Parallel data fetching: initiates all data requests at same time

Static rendering: data fetching and rendering happens on server @ build time or when revalidating data, useful for UI w/ **no data/data that is shared across users**, benefits:
- faster websites, prerendered content cached and globally distributed when deployed
- reduced server load, bc content cached, server doesn't have to dynamically generate content for each user request
- SEO, prerendered content easier for search engine crawlers to index

Dynamic rendering: content rendered on server for each user @ **request time** (when user visits page), benefits:
- real-time data, or data updated often
- user-specific content
- request time info, access info only be known @ request time, ie. cookies, URL search parameters
Note: w/ dynamic rendering **your app is only as fast as your slowest data fetch**

Streaming: data transfer technique that allows you to break down route into smaller "chunks" and progressively stream them from serve to client as they are ready
- can prevent slow data requests from blocking whole page


acme.com/dashboard
Partial content with loading state
Suspended content streaming in

Nextjs

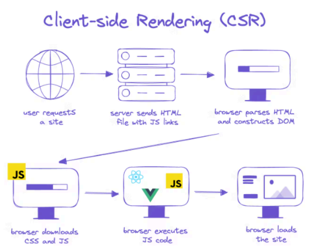
Next.js: open-source web dev framework that provides React-based web apps w/ server-side rendering and static rendering

Vercel: frontend cloud

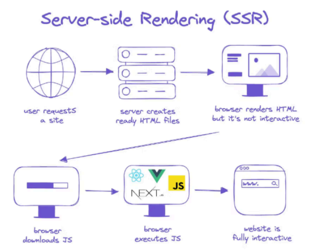
turbopack: bundling tool to improve performance

Tailwind CSS: open-source CSS framework, uses "utility" CSS classes

Client-side Rendering (CSR): generates HTML using js in browser


Client-side Rendering (CSR)
user requests a site
server sends HTML file with JS links
browser parses HTML and constructs DOM
JS
browser downloads CSS and JS
JS
browser executes JS code
browser loads the site

Server-side Rendering (SSR): generates HTML content on server, sends to client

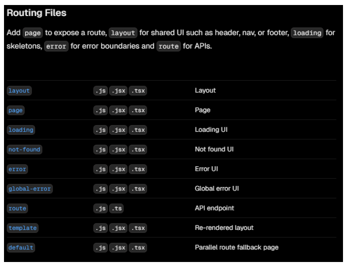

Server-side Rendering (SSR)
user requests a site
server creates ready HTML files
browser renders HTML but it's not interactive
browser downloads JS
NEXT.
JS
browser executes JS
www.
website is fully interactive

Quick Start

1. npx create-next-app@latest <app-name>
2. cd <app-name>
3. pnpm dev

Go to http://localhost:3000

Folder Structure
* /app: contains routes, components, logic for app
* /app/lib: contains functions for app, ie reusable utility and data fetching functs
* /app/ui: contains UI components for app
* /public: contains static assets for app, ie imgs
* config files: configuration for nextjs in project



Routing Files

Add `page` to expose a route, `layout` for shared UI such as header, nav, or footer, `loading` for skeletons, `error` for error boundaries and `route` for APIs.

`layout`	`.js` `.jsx` `.tsx`	Layout
`page`	`.js` `.jsx` `.tsx`	Page
`loading`	`.js` `.jsx` `.tsx`	Loading UI
`not-found`	`.js` `.jsx` `.tsx`	Not found UI
`error`	`.js` `.jsx` `.tsx`	Error UI
`global-error`	`.js` `.jsx` `.tsx`	Global error UI
`route`	`.js` `.ts`	API endpoint
`template`	`.js` `.jsx` `.tsx`	Re-rendered layout
`default`	`.js` `.jsx` `.tsx`	Parallel route fallback page

Types of Folders
_folderName: private folder
[folderName]: parameter of route
(folderName): route group

Tailwind: CSS framework that allows you to quickly write utility classes directly in React
* style elements by adding class names ie:

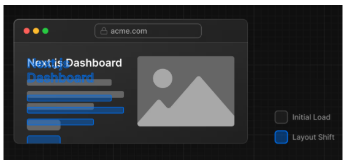
```
<h1 className="text-blue-500">I'm blue!</h1>
```

turns <h1> blue

* can either use tailwind or css modules

clsx: library that lets you toggle class names

Cumulative layout shift: Google metric to evaluate website performance and user experience, ie layout shifts happen when browser initially renders text in a fallback font then swaps it out for a custom font after loading causing elements to shift around



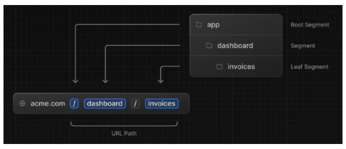
* **next/font** module downloads font files @ build time and hosts them w/ other static assets automatically optimizing fonts

* **anti-aliasing**: smoothing out edges in font by creating gradual transition, ie. tailwind has **antialiased** property

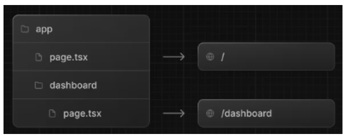
* **next/image**: **<Image>** component is an extension of HTML **<img>** tag, it automatically optimizes image by:
- preventing layout shift when imgs loading
- resizing imgs to avoid shipping large imgs to devices w/ smaller viewport
- lazy loading imgs by default (imgs load as they enter viewport)
- serving imgs in modern formats, ie WebP, AVIF, when browser supports it

* Note: define **<Image>** dimensions and fonts to prevent layout shifts

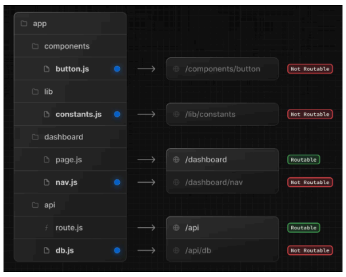
Nested Routing: folders used to create nested routes, each folder represents route segment that maps to URL segment



* **page.tsx**: components for pg, required to make route accessible

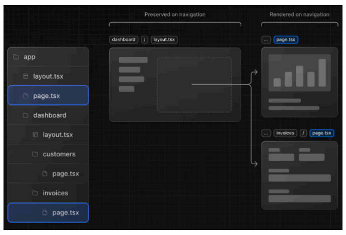


* can **colocate** publicly accessible and inaccessible files in app folder
* only content inside **page** file will be publicly accessible



* **layout.tsx**: root layout must contain <html>, <body> tags, changes affect all routes under folder, on navigation only pages components update while layout doesn't re-render

* **partial rendering**: preserves client-side react state in layout when transitioning between pages



* **root layout**: required in every nextjs app, UI shared across all pages in app

* **<Link />**: component allows client-side navigation w/ js, nextjs **prefetches** code for linked route in background

Note: nextjs automatically code splits app by route segments, pages are isolated, ie. if page throws error, rest of app still works, less code for browser to parse, makes app faster

* **seed**: populating a database w/ initial set of data

Note: don't query database directly when fetching data on client as exposes database secrets

React Server Components: benefits:
* support js Promises, providing solution for async tasks like data fetching natively, ie. can use **async/await** syntax w/out needing **useEffect, useState** or other data fetching libs
* run on server, keep expensive data fetches and logic on server, only sends result to client
* query database directly w/out additional API layer

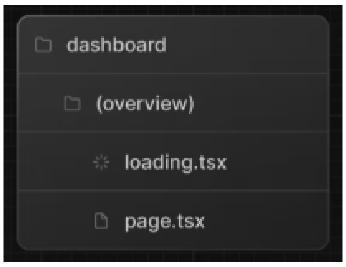
loading.tsx: fallback UI while page loads

* user doesn't have to wait for page to finish loading before navigating away called **interruptable navigation**

* **loading skeleton:** simplified version of UI as placeholder/fallback for loading content

Nextjs

Route Groups: ie. (folderName), doesn't affect URL path

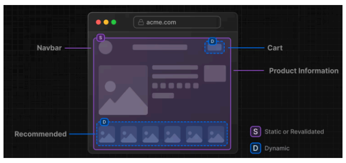


* ie. loading.tsx only applies to page.tsx file
* ie. /dashboard/(overview)/page.tsx becomes /dashboard

Suspense: allows you to defer rendering parts of app until some condition is met

Note: it's good practice to move data fetches down to components that need it, then wrap components in **Suspense**

Partial prerendering (PPR): combines static and dynamic rendering in same route, uses react Suspense, works by:
1. Static route **shell** is served immediately, makes initial load faster
2. Shell leaves holes where dynamic content loads asynchronously
3. Async holes streamed in parallel, reducing overall page load time



Note: if you call a **dynamic function** in a route (ie querying your database), the entire route becomes dynamic

Note: Suspense fallback is embedded into initial HTML file w/ static content, @ build time/revalidation static content is **prerendered** to create static shell, rendering of dynamic content **postponed** until user requests route
* Suspense is used as boundary btwn static and dynamic code

* **URL search params** benefits:
- bookmarkable and sharable URLs since search params in URL
- server-side rendering, URL parameters can be directly consumed on server to render initial state
- analytics and tracking, search queries and filters directly in URL makes easier to track user behavior w/out requiring additional client-side logic

Adding Search Functionality
*** useSearchParams**: allows you to access parameters of current URL, ie. search params for URL /dashboard/invoices?page=1&query=pending would look like: {page: '1', query: 'pending'}

* **usePathname**: lets you read current path, ie. /dashboard/invoices, usePathname would return '/dashboard/invoices'

* **useRouter**: enables navigation btwn routes w/in client components programmatically

Implementation steps:
1. Capture user's input
2. Update URL w/ search params
3. Keep URL in sync w/ input field
4. Update table to reflect search query

"use client": client component, can use event listeners and hooks

* **URLSearchParams**: Web API provides utility methods for manipulating URL query parameters

Pre-Populating
defaultValue vs **value**/ Controlled vs Uncontrolled
* if using state to manage input value, use **value** to make it a controlled component, React would manage input's state
* if not using state, can use **defaultValue**, means native input manages own state

useSearchParams() vs **search Params**
ie.
* **<Search>** is client component, use **useSearchParams()** hook to access params from client
* **<Table>** is server component that fetches own data, can pass **searchParams** prop from page component
Note: if you want to read params from client use **useSearchParams()** hook as avoids having to go back to server

Debouncing: programming practice that limits rate @ which function can fire, steps:
1. **Trigger Event**: when event that should be debounced occurs, timer starts
2. **Wait**: if new event occurs before timer expires, reset timer
3. **Execution**: timer reaches end of countdown, debounced function executed

Pagination: allows users to navigate thru different pages

React Server Actions: allow you to run async code directly on server, eliminate need to create API endpoints to mutate data, include features like:
- encrypted closures
- strict input checks
- error message hashing
- host restrictions
- etc
- advantage of invoking Server Action w/in Server Component is **progressive enhancement**: forms work even if js has not yet loaded on client

Dynamic route segments: when you don't know route segment names ahead of time and want to create routes from dynamic data, dynamic segments filled in @ request time/prerendered @ build time
Ie. **[foldername]**

Universally Unique Identifier (UUID): 128-bit number designed to be unique identifier, reduces risk of ID collision, globally unique, reduces risk of enumeration attacks, better for large databases

Auto-incrementing keys: ie (1, 2, 3, …), may cause ID collision in URL

error.tsx: can be used to define UI boundary for route segment, catch-all for unexpected errors and allows you to display fallback UI to users
notFound.tsx: error page for fetching resource that doesn't exist, takes precedence over **error.tsx**

Accessibility
Sematic HTML: using semantic elements (<input>, <option>, etc) instead of <div>, allows assistive technologies (ATs) to focus on input elements and provide context to user
Labelling: including <label> and htmlFor attribute for descriptive text
Focus Outline: fields properly styled
Form validation: either client/server-side validation for form

Authentication: checks who you are

Authorization: determines what you can do/access in app

Hashing: converts string into fixed-length string of characters which appears random, provides security if user's data exposed

Metadata: info embedded in page's HTML not visible to users, usually w/in <head> element, info is crucial for search engines/systems that need to understand webpage's content

Types of Metadata
* **Title**: title of webpage
* **Description**: brief overview of webpage content
* **Keyword**: keywords related to webpage content
* **Open Graph**: enhances webpage representation when shared on social media platforms
* **Favicon**: links favicon to webpage displayed in address bar/tab

2 Ways to Add Metadata
* **Config-based**: export static metadata object/dynamic generateMetaData function in layout.js /page.js file
* **File-based**: use nextjs special files for metadata

Note: can create dynamic OG images using ImageResponse Constructor
Note: metadata in nested pages will override metadata in parent

Nextjs

StackBlitz
https://stackblitz.com/edit/stackblitz-starter
s-j4zz7nvj