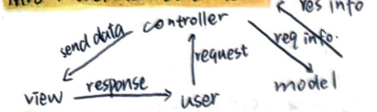
mvc model view controller



View: front end (HTML, CSS, client side JS)
Controller: API handler, Next.js framework (request handling, interaction with client, query db..)
model: Data management (define data, fields, store)
Data persistence: ... repetitive, high overhead, existing frame work or package done
Q2
Object Relational mapper: an abstraction over db queries
Simplicity, Consistency (JS), db and backend easy switch
Object oriented Programming, run secure & efficient query
method/attr. accesses are translated to queries | result/attr. wrapped by objects/ attr. SQLite: lightweight, dev DB
Models: represents & manage application's data (M)
defined as classes —ORM map→ table in DB
ORM = Prisma generate JS classes from its schema file
install: npm i prisma @prisma/client @prisma/studio
Run: npx prisma init (create schema.prisma)
generate relevant JS class npx prisma generate → JS
syc schema with DB npx prisma migrate dev →
view dB: npx prisma studio Access from localhost:5555
Field attr. @id @default(autoincrement()). unique, map, index
Types: Int, String?, Boolean, BigInt, Float, Decimal, DateTime, Json..
relation [categoryId Int [categoryId], references: [id])
category Category @relation("CategoryProduct", fields: ↑
product Product[] @relation("categoryProduct") @只加在Fkey那边
Book[] ×不加 (many)
1 to 1: set 2边的 foreign key @unique
many to many relation R[] at each end @@id(id1, id2) 除非给relation 起名]
model R(Id1 t1 Table1 @relation t2 ...)
import {PrismaClient} from "@prisma/client";
export const prisma = new prismaClient();
findFirst / Unique / UniqueOrThrow
Select: const products = await prisma.product.findMany({ GET
where = { name: name? {contains: req.query.productName, },
(filter) price: {lt: 3000, }, is_available: true, },
(object) include: { store: true, }, }) POST connect: {id: 1}, }
Insert: prisma.post.create({ data: {title: "t1", user: {↑ }});
prisma.post.update({where: {id: 'id1'}, connectOrCreat PUT
data: { password: "new"}, }) updateMany. DeleteMany
await prisma.product.delete({where: {id: 1}, }) DELETE
CRUD: created, read, Update, Delete eg RT DB opt
validation: requied field exists, field type match, specific format
Return 400 response for malicious / in valid data ↓before exe.q.
O/w: Security Vulnerability: may stored somehow
用 try catch. Number(id) parseInt(id) ? {contain: } : undefine.
的cascade 先查再删 是以区
Authentication: obtain user information
Session Auth: client sends user/pass at login → random server create session id
→ returned in response, browser saved in cookies, resuse
• less scalable (server store all sessions) & addi data query / req
• more control (server can revoke a session)
Token Auth: contain info about user, can be json string
• must be signed by server to avoid attacks (seemly random str)
• Simpler (no DB interaction) • more scalable (client store token)
less control (true logout impossible)
Best Practice: Toke is preffered; • Json Web Token (JWT)
main risk: compromised tokens (should send over a over, nonrevok)

• short live (expire) • frequent Auth is bad UX • refresh Token
Refresh Token: at login, client receive token & refresh token
Token is short live • use refresh token to get new token
• refresh token much longer • 401 Unauthorized (try refresh token first, then resend req with new acc token) • session continuity (only re auth when refresh token expires)
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";
const BCRYPT_SALT_ROUNDS, JWT_SECRET, JWT_EXPIRES_IN = ...
export async function hashPassword(Password){
return await bcrypt.hash(Password, SALT);}
export async function comparePassword(password, hash){
return await bcrypt.compare(password, hash); }
export function generateToken(obj){ Process.env.
return jwt.sign(obj, JWT_SECRET, expiresIn: JWT_EXPIRES_IN, }); }
export function verifyToken(token){ req.headers.authorization
if(!token?.startsWith("Bearer ")){return null;}
token = token.split(" ")[1]; process.env.
try { return Jwt.verify(token, JWT_SECRET); }
catch(error){ return null; } }
const user = verifyToken(req.header.authorization);
if (!user) {return res.status(401).json({message: "Unau"})};
return.status(200).json({message: `Hello ${user.name}`});
create(... password: await hashPassword(password)
Authorization: check user's properties & permission (权限)
check before API handler, access? 403 Unauthorized
reusable logic, seperate from handler in middlewares
Migration database tabe same state model schema (independent)
ORM: apply schema to Database (via DDL query) mig folder.
self-sufficient, don't rely on db History of changes stored in
• not contact database 2. Iterate over all models to find diff
1. builds the old db state from previous migration 3. contain DDL query new folder
• a migrate should not be applied twice!
• migrations themselves are stored in database.
migration table: migrations stored in _prisma_migration Table
• metadata (content is in migration file)
• checksum ensures migrations are not edited
Generate new m file: Build temp schema from all existing m files → compare schema w. models → generate diff as new m file
Apply m to DB: query _table and list all ms, List all sub under m folder
→ Find m that are not present in table → for each m: exe m.sql file
→ insert migration info into _prisma_migrations table
assum: never change m & DB; m are same everywhere (push to git)
Select: npx prisma migrate deploy (Applied unapplied m (without create new one)
error: npx prisma migrate resolve --applied "migration_name" or --roll back "migration_name": will only update migration table, without execute querys
• manually sync database schema with migrations
npx prisma db reset: delete all table's data, apply m on empty db
role: role # "USER" (default) await 加在func前(调用)
const payload = {username: .., role: ..} 定义时 加 async
obj = {...payload, expiresAt: Math.floor(Date.now()/1000)+60x15}, //15min
{expiresIn: '15m'}
middleware:
export function RequireAdmin(req){
const user=verifyToken(req.headers.authorization);
if (user.role != "Admin"){
throw new Error('Forbidden'); }
return user; }
export default async function handler(req, res){
try{ const user= requireAdmin(req);
res.status(200).json({message: 'Yes'}); }
catch(error){if error.message === 'Forbiden'){...}
Classic Web Application backend listen HTTP req
req from browser (a page) server return HTML Body
Browser renders HTML page

Morden: req from browser, app, postman... CRUD opt
return JSON Body. use JavaScript to make changes to the web page. called single page Application (no full reload)
single page application. Seamless user experience (No reload no refreshes). doesn't get reset everytime. more control UX)
• Efficiency (the whole page don't get updated) • Faster load time (the initial load when nothing is there) takes less time.
Asynchronous JavaScript and XML (Ajax): browser send the request in backend (don't block main thread)
web server → XML or HTML or JS data → Ajax engine → HTML CSS → UI (web browser)
Q4 & backend front end seperation.
React (Ajax Frame work) JS library for build Interactive UI
• re-render (no longer need manipulate ele manually)
• creat virtual DOM in mem • changes → rerender
• compare new and old DOM → what's update • update specific elements of the browser's DOM (只改变了的)
JSX variation of JS, merging HTML and JS together.
browser don't understand (tranblate before execution)
component (make ele reusable), funct/class return ele (React) like Tag
function Say(){
return <h1>Hello</h1> } React component → DOM eg <div> container
reuse: ReactDOM.render(<Say/>, document.getElementById("root"));
component's name capitalized, lower case for built in ele eg p, h1
Tag: < /> more than one: <> ... </>
props: react mimic JS attr. via props (readonly data coming from the parent element
Dictionary: function Text(Props){ return <h4>{props.value}</h4>}
To pass props: <Text value="Joh" />
style and classes JS statement {} inside JSX
function Text(props){
return(
<h4 className="Text" style={{front size=prop.size}}> (Convert to JS; JS object)
{props.value}
</h4>)} passprops <Text value="car", size={30} />
function List({title, values}){ (destruction 拆开props)
return(Component name (call 处)
<> <Text value={title} size={40} />
 {values.map((item, index) => (
<li key={index}> {item}))}
 element created in a lop must have unique
</>)} key prop ⇒ identifies which item Changed otherwise, rerender whole list when sth changes
paired tag
use component as paired tag
function Wrapper({children}){
return <div className="col"> {children}</div>;
const wrapped=(
<wrapper> pass as Children prop
<List values={[1,2,3,"my cat"]} />
<wrapper/>)
rerendering and updates
Class components. define a component
Class Welcome extends React.Component{
render(){ props passed to constructor
return <h1>hello, {this.props.name}</h1>; } }
state Initialize the state object in the constructor
Class Counter extends React.component{
constructor(props){
☆ super(props) never assign state other than rerender
this.state = {counter: 0}} }
render(){ state values be accessed via this.state
return <h3>{this.state.counter}</h3> } (other won't rerender)
react states should never be muted update setState

update state this.setState({counter: this.state.count +1})
state same set of events as JS. camelCase
• The action must be a function, not any statement
eg onClick{() => alert()} vs onclick="alert()" not function
deal with this ← evaluated on loading of ele; will call imme
event handler
increment(){this.setState({counter: this.state.counter +1})}
use: <button onclick={this.increment}> click me </button>
Each JS func has its own this (caller object) dom obj!
The object that call the event handler is not your→
Sol: constructor(){this.onclick = this.onclick.bind(this)} (counter; original method; counter)
Sol: use arrow func (don't introduce own this, capture this from outer scope, the class has proper this)
<input incre=() => {this.setstate(..)}
type="text" read from state
value={this.state.celsius} Store to state celsius: event.target.value}
onchange={event => this.setState({...this.state, current input value.
value变了就exe 并且pass the event object as its first parameter "event" obj
class Input Extends React.component{
constructor(props){ super(props);}.
handle change = (event) => { this.props.onChange(event.target.value);} send update value to parent call parent's handle change function
render(){ return(<div>
<label> {this.props.title </label>
<input type="text" value={this.prop.value}
onchang={this.handleChange} /> </div>)}}
class Converter extends Reate component{
constructor(props){super(props) this.state={celcius: 0 fahre: 32}
handle change = (isCelsius) => (value) => { (value*9/5+32)};
if (isCelsius){this.setstate({celcius: value, fahre:
else{ this.setState({fah=value) celcius: (v-32)*5/9);}}
render(){return(<div>
<Input title="Celsius" Value={this.state.celcius}
onChange={this.handleChange(true)} />
<Input title="fah" value={this.state.fah}
onChange = {this.handleChange(false)} /> </div>)}}
shared state in Ancestor (defined)
pass it as props to original component
Pass a setter func as change handler
JSX code be translated to JS every time => slow
React in Node.js: front end code, a series of HTML, CSS and JS files are returned. compiled JS component
• A pre compiled and bundled build for production
Monorepo: the practice of have all code in one repo
• Gaint code bases • don't deal with repo version
• share type, utils, library • project is self contained & easy to navigate • diff Node projects in 1 Parent Node Proj
Next.js: monolith: all code in one node project
• good for small pros • may migrate to mono if bigger
• Extract reused parts into seperate Node projects
react in Next.js • Create the HTML and compiles JSX
• You can import all your styles and assets to your JS module Handled and served properly by the server
• only program language need: JS!
File structure: import "@/styles/globals.css"
• Image & static gather under public

Hooks • Syntax sugars • ~~class, constructor, set state~~
- move back to function components.

useState State not a one object anymore
- Define seporate state variables via use state hook

```
Import React, {useState} from 'react';
```

- Return variables and updated function
- components gets rerendered when the value change

Benefit: • functions instead of verbose class components
- enable multiple state variables • no more this, method binding
- easy to share state with child elements (each state variable comes with its own Setter) 生成 (first tim creat or add)

Lifecycle: In class components: componentWillMount()
componentDidMount(); componentWillUnmount()
控制什么时候 run code (now run when render is called)

useEffect • a hook to replace life cycle functions
- called when component mounts • called when sth changed

```
useEffect(()=>{console.log("pros size or status changed")},
  [status, props.length]) any element of arry change, invoked.
```

- Do not leave out 2nd argument ⇒ run at every rerender inefficient! • `useEffect(()=>{console.log("s")}, [])`

run only once after initial ← when commonts

components/Input/Index:

```
import React from "react";
export default function Input({title, value, onChange}){
  const handleChange = (event) => { onChange(event.target.value);};
  return ( <div style={{fontFamily: "Arial"}}>
      <label> {title} </lable>
<input type="text" value={value} onChange={handleChange}/>
      </div> ); }
```

Converter:

```
const [celsius, setCelsius] = setState(0);   // initial
const [fah, setFah] = setState(32);
useEffect(()=>{console.log("useEffect called);},[]);
const handlechange = (isCelsius) => value => {
  if (isCelsius){ setCelsius(value);
                  setFah((valuex9)/5+32); ]
  else {..} };
return (
  <div>                   onChang(handleChange(true)] />
    <Input title="celsius" value={celsius} ↗
    <Input...
  </div> ); }
```

Fetch API The interface for browsers to resend HTTP req (native support for REST framework)
Returns a Promise

```
Const [holiday, setHoliday] = useState([]);
const fetchHolidays = async () => {
  const response = await fetch("https://web/api/v1/holidays");
  const data = await response.json();
  setHolidays(data.holidays"[]); }
useEffect(() => {fetchHoliday()}, [])
```

Pagenation

API authentication **First-party auth**: store token in client's persistent storage
- should not be deleted when Tab/browsor (关闭) is closed

Web browser: use localstorage

```
localStorage.setItem('accesse_token', access_token);
localStorage.getItem('access_token');
```

- set Authorization header with appr value.

Third Party: API keys (permanant or very long

CROS Cross-Origin Resource Sharing
- A client only request to URLs with same Domain
- Browser block you from fetching a different domain
- API server also block requests coming from server

sol: **backend Proxy** • Implement a backend API that requests the third party service and return res

Pros: • API key is not exposed • more control over what data is transferred / who access data
- Better logging and monitoring

eg (sign in use Google) ← redirect
- Server contacts Google with auth code and API key to receiver user info. • server create acc, generate token

UseRouter manage & update query parameter dynamicaly,

```
const router = useRouter();        // default
  ↳ let you access query para from URL
router.query.page
useEffect(() => { Const queryPage = ParseInt(^,10)||1,
setCurrentPage(queryPage)}, [router.query.year]);
```

Text: e.target.value
↳ event.

```
const handleProvinceChange = (e) => {const new = e.target.value;   → onChange={handleProvinceChange}
  setProvince(new);  --→ useEffect(()=>{..}, [Province])
  router.push({
      pathname: "/"
      query: {...router.query, province=new, page:1},});};
const pagedate = useMemo(() => {
  const startI = (current Page - 1) * Holiday Per page
  Const endI = start I + Perpage
  return filteredHoliday.slice(sI, eI);}
  [current page, filteredHolidays]};
const handlenextPage = () => { next page
                               setcurrent Page
                               Push }
```

Arguments
[id].jsx `const {storeId} = router.query.`

Links: `<Link href="/watch"> watch </Link>`
Link & Router from Next, not react.

context React's way to handle global state.
accessible within its provider.

```
export const TextContent = CreateContext({
    Var1: nul, var2: null, });
```

'put default value ☆

```
create myobject = {var1: 1, var2: 2};
<TestContext.Provider value={myobject}>     Parent
  ...       可以放 useState
</TestContext.Provider>
```

At any descendent, you can access

```
const {var1, var2} = useContext(TexContent)
```

- good to used by many components / diff com
- create context for each set of relevant variables & their setters

type safety **Static Typing**: check at compile time
JS: → Dynamic: Check at runtime (type can Δ)
Strong: strict rules
Week: flxible with type conversion, → type coercion

Typescript: superset of JS, no runtime effect
Pros: Improve code quality, catch error during deployment, not at runtime • Better colla, tooling, auto Completion
checks at compile time, RT code is JS

Type declaration:

```
let message: string = "Hello!"
function greet(name: string): string {
    return `Hello, ${name}` }
```

Type Inference `let count = 45`

Type System Primitive types: arr Tuple
- JS types • any, unknow, void, never • number[]
  - Emum • optional name?: string [number, string]

Type aliase: `type ID = string | number` **generic**

Interface

```
ModalProps {                  | const Modal: React.Fc(ModalProps)
  text: String                |   = (props) => { const [loading, setloading]
  image?: string }            |   = useState<boolean>(false)}
```

@ts-ignore

check at RT: typeof instance of

Advanced CSS **Traditional CSS** (css bloat (unsed style), Specificity war (complex, dependence)
CSS-framework (早-) contex switch btw JS, CSS
Replace CSS with utility class

```
<button className="bg-blue-500 text-white
    font-bold py-2 px-4 rounded">
  Click me </button>
```

`npm install tailwindcss postcss`
- arbitrary values "w-[50%] text-[#ff6347]"
  - Dark and white modes "bg-white dark:bg-gray-800"
  - responsive styles "p-4 sm:p-6 md:p-8" (padding)
- CSS interoperability `h1 {@apply text-2xl font-bold;}`
- customize themes

resposive design: well in diff devices | rem = 4
tailwind translate to rem:

Flex horizontal or vertical places items inside the parent ele (container)
- add class = "flex" eg "flex flex-col" to container
- To wrap items on overflow "flex-wrap"
- spacing "justify-center / between / around / evenly"

Flex item: Control how much space each item take btw
- flex-1 to take whatever space left. • Divid it mul

Gridlayout: in container
- class = "grid grid-cols-3 gap-4"
- how much space each item need class = "col-span-2"
- responsive class = "grid sm:grid-cols-2 md:grid-cols-3 gap-4"

X-horizontal y: vertical P: padding M: margin
Note: sm, md: grid-cols-2
- class Name blot • reused class → new com
- use @apply to move the classes to CSS

shallow mb: margin bottom hover.

Q:
Isolation virtual machine: Full isolation, heavy
Sandboxing-limit process's access to resources
chroot jail (restrict access)

Docker: • dvelop, ship, run app • allow you to package an app with all of its dependencies into a std unit call container • portable: stopped, restarted and copied easily

Pros: consistancy across env • simplified dependency management
- containers run in isolation • Easier continuous Integration and deployment (CI/CD)
- shared kernal space, isolated user space
- low performance overhead

```
docker (run) -it python:2.7   "interative"  /bin/bash
Dockerfile                              Dvelopment
From node:20-alpine
WORKDIR /app
RUN npx --yes create-next-app --yes
WORKDIR nextapp
COPY test.tsx app/page.tsx
RUN npm install
EXPOSE 3000
CMD npm run dev
docker build --network=host -t next.
  -f next.Dockerfile
docker run -p 80:3000 next
          ↓ local   ↳ container
```

Docker Image:
- lightweight, standalone, executable package
- read only templete to create Docker container
- Build start from a base image. • built in layers (cached)
- immutable, portable • visioned → tag (-t)

Docker container: instantiate from image
docker ps, stop, log

Docker volume: persistent data storage, sharing data between container (eg data base, HTTP certificate)

```
docker volume create pgdata
docker run -d --name mypg \
-e POSTGRE_PASSWORD=password \ -v (path)
-p 5432:5432     run on start up
```

Dock compose Specify an order of container to be
- docker-compose.yml
- docker-compose up (down)

Deployment
Server: Computer, static public IP

Deve vs Deploy (error: generic, run with real web server)
(test db, error: actual trace, run with dev server)
npm run dev: ① performance: compile when you requ & hot reload.
② environment diff should eg db URL
⇒ env.production NEXT_PUBLIC...
- env × on git venv.sample

production build: ~~npm run build~~
- compiles and bundle all files
- Minifies JS, and perform several optimization
- much faster • in .next (pure JS, no TS, TSX)
- contain (server, static, cache/) npm start

npm install → prisma generate → build
no migratio: self contained × introduce dependency code
no access to DB

Process Manager lifecycle (restart... run continu)
- Scaling the app (# worker, core) • Monitoring (log, metris)

```
npm install -g pm2
pm2 start "npm start" --name nextapp
daemonize: running in back ground
  pm2 startup     pm2 save, list, logs name
                  restart name
```

why copy? something don't need
① code: be already built all code pages directories packages except packet json and the public
② dependency = not used in prod. eg typescript
pm2-runtime? pm2 run in background
CMD cannot be background process.

Web server Nginx symbolic link expose to Int
Static files: files / dir granted to webserver
static JS, CSS, HTML, served by web server itself & perm