Tema 4

NextJS

NextJS es un framework basado en React que nos permite crear páginas web rápidas y de manera sencilla. Además esta nos permite usar nodejs para usar los paquetes que necesitemos.

Lo interesante de usar NextJS es que permite compilar y renderizar paginas webs en el lado del servidor, lo que hace que si no hay cambios en la página, cada vez que alguien nos pida la pagina se lo enviemos de manera rápida y el cliente la cargue rápidamente.

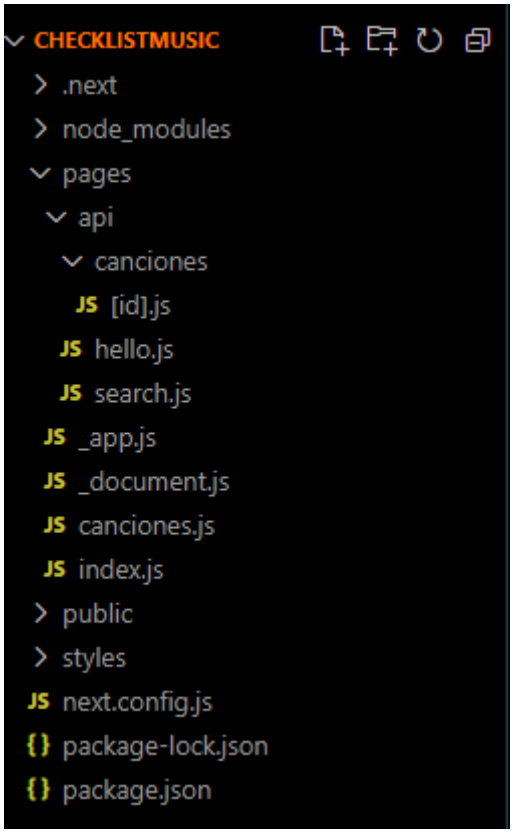
En mi caso, he realizado una aplicación bastante simple de creación de checklist para grupos de música, es decir, puedes buscar un grupo cualquiera y que te cree una lista con todas sus canciones para que puedas ir tachando las ya escuchadas y escuchar su discografía completa.

He realizado una implementación muy simple a la que le faltarían bastante features de una aplicación completa pero que es perfectamente valida para mostrar las distintas cosas que son posibles con NextJS. Empezando por la creación del proyecto, donde tenemos un par de opciones, la de hacerlo de manera manual o automática. Para hacerlo de manera automática simplemente tendríamos que correr en una terminal:

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

Esto nos llevara a través de todos los pasos para crear la app, dándonos opciones como el nombre o si queremos usar typescript.

Tras esto, nos creara una carpeta con la siguiente forma:



Donde tendremos en .next y node_modules, archivos que usa nextJS para funcionar y los módulos de nodeJS. En pages colocaremos los distintos archivos que queremos que sean páginas, aunque podemos crear mas carpetas dentro de esta. Dentro de pages, vemos la carpeta api, aquí pondremos las distintas APIs que vamos a usar para conseguir datos.

En public podremos colocar cosas como imágenes y en styles los estilos.

Como se puede ver, tanto paginas como las APIs, son archivos de JavaScript. Esto es debido a como funciona NextJS, a través de funciones, por lo que si vemos alguna pagina como "_document.js" que es una de ejemplo vemos:

```
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html lang="en">
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

Donde creamos una función default, con esto hacemos que cuando carguemos la dirección(que será la dirección del archivo dentro de pages) lance esa función por defecto. Así que en return pondremos los elementos HTML que queremos que sean parte de lo que queremos ver. Lo interesante, es que podemos crear otras funciones y usarlas como elementos de HTML. Si creáramos una función como esta:

```
function helloWorld(){
  return (
    <p>Hello world</p>
  )
}
```

Podríamos introducirla en el default de la siguiente manera:

```jsx
export default function Document() {
  return (
    <Html lang="en">
      <Head />
      <body>
        <Main />
        <helloWorld/>
        <NextScript />
      </body>
    </Html>
  )
}
```

Y al cargar la página, esa parte se sustituiría por el return de la función. Además, se le pueden pasar parámetros como cualquier otra función.

Las APIs también funcionan de manera bastante interesante aquí. Podemos definirlas de dos maneras, con una ruta fija o dinámica. Una fija podría ser el api “hello.js” que se vería así:

```jsx
export default function handler(req, res) {
  res.status(200).json({ name: 'John Doe' })
}
```

Y podríamos acceder a ella poniendo simplemente “/api/hello”. Pero, NextJS nos permite llamarlas de manera que podamos acceder al valor directamente, por ejemplo, podríamos renombrar el api a “[nombre].js”, de esta manera la haríamos dinámica. Así que si queremos acceder a ella, ya no tenemos que poner “api/hello”, ahora podemos poner cosas como “api/manolo”, lo que hace que le pasemos el valor “manolo” al api. Si queremos usarlo, tendríamos que hacerlo de la siguiente manera:

```jsx
export default function handler(req, res) {
  res.status(200).json({ name: req.query.name })
}
```

Explicado el funcionamiento de NextJS, voy a empezar a explicar mi app. Cuando cargamos la url base, vamos a la página index la cual sería:

```jsx
import { useState } from "react";
import Link from "next/link";
export default function HomePage(){
    const [result, setResult]=useState(null);
    const [loading, setLoading]=useState(null);
    const [error, setError]=useState(null);
    const [formData, setFormData]=useState({nombre:""});
```

Primero definimos vario estados, lo cual nos permite crear ciertas variables y una manera de actualizarlos. A useState le pasamos como parámetro, el valor inicial del estado.

```
const handleSubmit = async (event) => {
    event.preventDefault();
    setFormData(event.target[0].value);
    setLoading(true);
    setError(null);

    try {
        const res = await fetch('/api/search', {
            method:'POST',
            body: JSON.stringify(formData),
            headers: {
                'Content-Type':'application/json',
            },
        });
        const data = await res.json();
        console.log(data);
        setResult(data);
    }catch(error){
        setError(error);
    }
    setLoading(false);
};
```

Tras esto, tenemos la siguiente función asíncrona. Esta se lanza al hacer submit del formulario que veremos mas abajo. Primero usamos preventDefault(), para que no se envía el formulario.

Conseguimos el valor del formulario y lo guardamos en FormData, ponemos la variable loading a true y el error a null. Y entramos en un try catch, el cual lo que hace es llamar a la api como post, pasándole el valor del formulario como body. Tras esto, en data esperamos el resultado y nos quedamos con el json. Por último, guardamos el resultado en el state result y ponemos la variable loading a false.

```
return (
    <form onSubmit={handleSubmit}>
        <input type="text" id="nombre" name="nombre" />
        <button type="submit">Buscar</button>
        {loading && (<p>Cargando</p>)}
        {!loading && result &&
            <div>
                <table>
                    <thead>
                    <tr>
                        <th>Nombre</th>
                        <th>Pais</th>
                        <th>Descripcion</th>
                    </tr>
                    </thead>
                    <tbody>
                    {result.map((item)=>{
                    return(<tr>
                        <td><Link href={{pathname:'/canciones', query:{id:item.id, nombre:item.name}}}>{item.name}</Link></td>
                        <td>{item.country}</td>
                        <td>{item.disambiguation}</td>
                    </tr>)
                    })}
                    </tbody>
                </table>
            </div>}
    </form>
)
```

Ahora tenemos el return de la función, la cual de base nos mostrará solo un formulario con un campo de texto y un botón submit. Tras esto nos encontramos con:

```
{loading && (<p>Cargando</p>)}
```

De esta manera, podemos hacer que cuando enviemos una request al api, mientras cargan los datos, se muestre por pantalla la palabra Cargando.

Tras esto tenemos:

```
{!loading && result &&
    <div>
        <table>
            <thead>
            <tr>
                <th>Nombre</th>
                <th>Pais</th>
                <th>Descripcion</th>
            </tr>
            </thead>
            <tbody>
            {result.map((item)=>{
            return(<tr>
                <td><Link href={{pathname:'/canciones', query:{id:item.id, nombre:item.name}}}>{item.name}</Link></td>
                <td>{item.country}</td>
                <td>{item.disambiguation}</td>
            </tr>)
            })}
            </tbody>
        </table>
    </div>}
```

Esto se nos mostraría si tenemos un resultado y no esta cargando. Seria una tabla con el nombre, país y descripción de la banda que tenemos de resultado

Y en el cuerpo tenemos una función map, la cual nos permite iterar sobre los elementos del json de resultado y crear la tabla con cada uno de ellos. Como podemos ver, lo primero que hago es crear un enlace a otra página, donde le paso como query el nombre del grupo y el id interno para luego poder usarlo.

La siguiente pagina que vamos a ver es canciones, que es a donde nos lleva si pulsamos el nombre de un grupo en la tabla. Empieza de la siguiente manera:

```
export default function Home() {
    const router = useRouter();
    const id = router.query.id;
    const [names, setNames] = useState([]);
    const [struckOutNames, setStruckOutName] = useState(new Set());
```

Donde tenemos un router, que esto lo que hace es que podamos leer las querys de los usuarios, además de más información. Y creamos dos estados, uno con una lista vacía, y otra con un set. En la de nombres guardaremos los nombres de todas las canciones y en la segunda, los nombres que ya hemos tachado.

Ahora tenemos:

```
useEffect(() => {
  async function fetchNames(){
    try{
        const res = await fetch(`/api/canciones/${id}`, {headers:{Accept:"application/json"}})
        const data = await res.json();
        const titulos = [];
        data.map((item)=>{
            titulos.push(item.title)
        })
        setNames(titulos)
    }catch(error){}
  }
  fetchNames()
}, [`/api/canciones/${id}`])
```

useEffect es algo que nos permite ejecutar trozos de código dependiendo del estado de la página. En este caso lo usaremos para conseguir los nombres de las canciones. Creamos la función asíncrona fetchNames la cual hará una llamada a nuestro api con el id del grupo. Esperara el resultado y guardara uno a uno los nombres de cada canción. Cuando termine, los guardara en el state names que hemos definido al principio. Colocando al final lo que hemos puesto, evitamos que entre en un bucle infinito la página, ya que llamara a la api, y dejara de volver a renderizar cuando los valores de name sean como los del final de useEffect.

```
return(
    <div>
        <h1>{router.query.nombre}</h1>
        <table>
            <tbody>
                {names.map((name)=>(
                    <tr key={name} style={struckOutNames.has(name) ? {textDecoration:'line-through'}:{}}>
                        <td>
                            <button
                            onClick={()=>{
                                setStruckOutName((prev)=>{
                                    const newSet= new Set(prev);
                                    newSet.add(name);
                                    return newSet;
                                });
                            }}>
                                Tachar
                            </button>
                        </td>
                        <td>{name}</td>

                    </tr>
                ))}
            </tbody>
        </table>
    </div>
)
```

Ahora tenemos el return, donde primero mostramos el nombre del grupo y creamos una tabla. Esta contendrá el nombre de cada canción y un botón para tacharla. Usaremos la función map para iterar sobre la lista.

```
<tr key={name} style={struckOutNames.has(name) ? {textDecoration:'line-through'}:{}}>
```

Cada fila tendrá una key, y el estilo de texto cambiara dependiendo de si el nombre del canción esta en el set que hemos creado al principio.

```jsx
<button
onClick={()=>{
    setStruckOutName((prev)=>{
        const newSet= new Set(prev);
        newSet.add(name);
        return newSet;
    });
}}>
    Tachar
</button>
```

El botón contendrá la siguiente función, la cual tomará el set actual, añadirá el nombre de la canción que hemos tachado y lo devolverá. Y añadiendo el nombre en cada fila, tenemos la tabla completa.

Ahora voy a explicar las APIs que he creado

Primero la de buscar grupos que sería search.hs:

```jsx
export default async function handler(req, res){
    let valor = req.body;
    valor = valor.replace(" ", "%20");
    const resultado = await fetch("https://musicbrainz.org/ws/2/artist/?query=artist:"+valor, {headers:{Accept:"application/json"}})
    const data = await resultado.json()

    res.status(200).json(data.artists)
}
```

Le pasamos en el cuerpo de la request el nombre del grupo y sustituimos los espacios por “%20” para que no se problemas. Tras hacer la request, esperamos los datos y los devolvemos.

Ahora [id].js, la cual usamos para buscar las canciones de cada grupo:

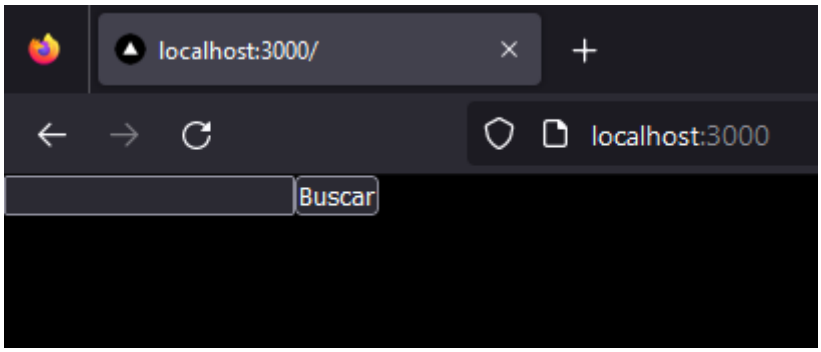
```jsx
export default async function handler(req, res){
    let url = "https://musicbrainz.org/ws/2/work?artist="+req.query.id+"&fmt=json&limit=100"
    const resultado = await fetch(url, {headers:{Accept:"application/json"}})
    const data = await resultado.json()
    var canciones = data.works
    const count = data["work-count"]
    var offset = 100
    while(offset<count){
        url = "https://musicbrainz.org/ws/2/work?artist="+req.query.id+"&fmt=json&limit=100&offset="+offset
        let resultado2 = await fetch(url, {headers:{Accept:"application/json"}})
        let data2 = await resultado2.json()
        canciones = canciones.concat(data2.works)
        offset+=100
    }
    res.status(200).json(canciones)
}
```

Para llamar a esta, se puede hacer de forma dinámica al haberla definido con corchetes. En esta primero creamos la url que vamos a usar con el id del grupo y llamamos a la api. Guardamos las canciones en una variable aparte, y guardamos el numero de canciones total que tiene el grupo. Esta api externa tiene un limite de 100 canciones por llamada, así que tenemos que usar un bucle while, el cual comprueba si el offset(El cual podemos usar para conseguir el resto de canciones) es menor a la cantidad total de canciones, en ese caso se ejecuta el bucle. Dentro de este volvemos a llamar a la api con un offset mayor y añadimos las canciones a la lista. Además de aumentar el offset en 100. Por último, devolvemos las canciones.

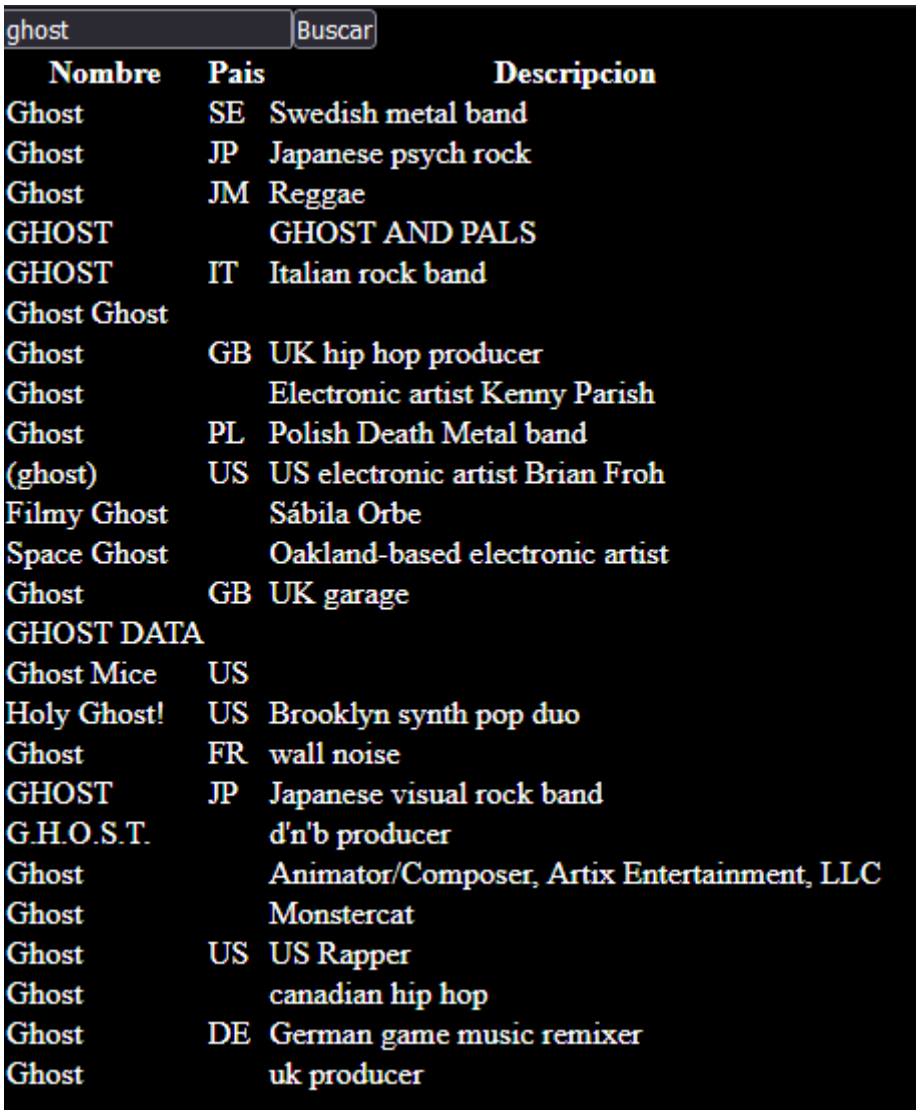
Ahora vamos a ver cómo hacer funcionar nextJS. Esto es tan simple como abrir una terminal e ir a la carpeta. Dentro de esta ponemos “npm run dev” y se iniciara el servidor:

```
PS C:\Users\david\Desktop\uni\desSerTel\tema4\checklistmusic> npm run dev

> checklistmusic@0.1.0 dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
event - compiled client and server successfully in 2.1s (166 modules)
```

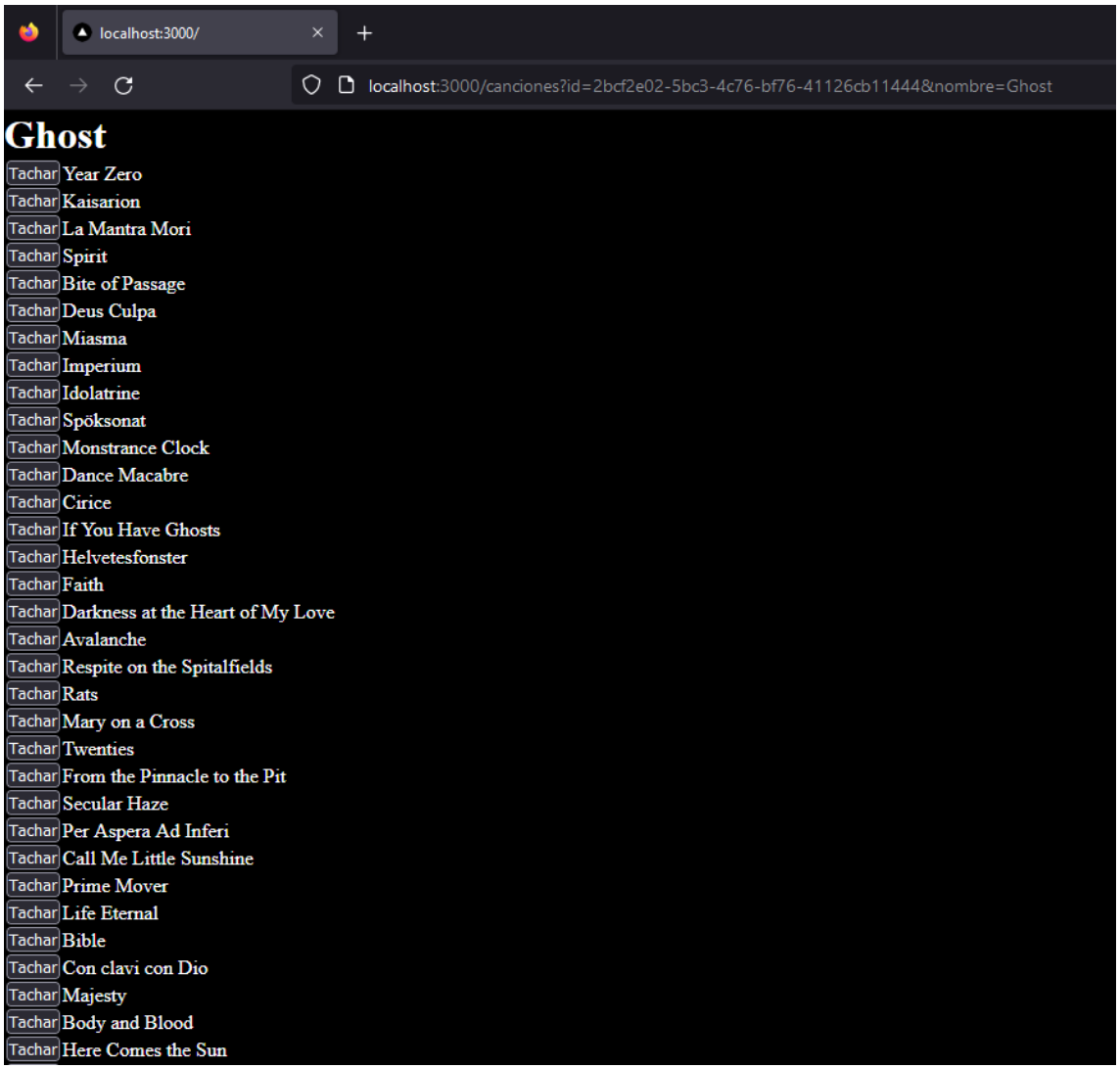
Ahora nos dirigimos a la url que nos dice y veremos lo siguiente:



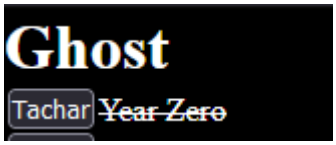
Si ponemos el nombre de algún grupo podemos ver:



Y si clickamos en alguno, como en el primero veríamos lo siguiente:



Si hacemos click en alguna canción pasa lo siguiente:



Como se ha podido ver, con NextJS se pueden crear aplicaciones relativamente fácil con un poco de aprendizaje. Y yo no he usado todo lo que te permite NextJS de base, hay cosas como sesiones para poder guardar el inicio de sesión de una persona o datos necesarios, poder llamar a una función que llame a APIs o consiga los datos de la sesión antes de cargar la página y así tenerlos listos desde el principio, etc. Además de permitirte usar React y nodeJS con lo que puedes hacer prácticamente lo que quieras con él.

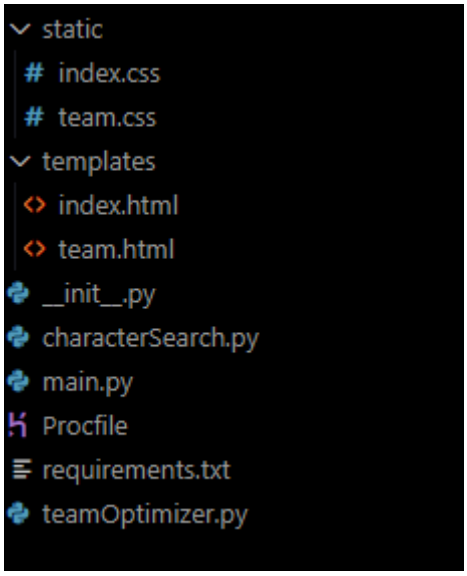
FastAPI

FastAPI es un framework de Python que nos permite hacer web app de manera rápida y simple, aunque permite complicar la cosa un poco usando distintas tecnologías.

En mi caso, hice una aplicación para terminal que me permitía introducir un equipo de un videojuego y me devolvía el mejor orden para el equipo. Esto lo hacia haciendo llamadas a una

API externa para conseguir la información de los personajes y con algo de código en Python procesarlo.

La estructura de carpetas tendrá la siguiente forma:



Donde en static pondremos elementos estáticos como css o imágenes y en templates los elementos de HTML. También se podrían hacer carpetas que funcionen como los módulos de Python para el código.

Los archivos characterSearch y teamOptimizer son los archivos que se encargan del backend, para usarlos con FastAPI no tuve que hacer ninguna modificación a estos.

Voy a hablar un poco de ellos para que se vea que hacen y quede claro luego el funcionamiento de la webapp. Empezando por characterSearch:

```python
class characterSearch:
    def __init__(self):
        self.url = "https://dokkan.fyi/characters/"
        self.awakening = ["Super", "Extreme"]
        self.type = ["AGL", "TEQ", "INT", "STR", "PHY"]
```

Es una clase cuyo constructor está vacío y solo tiene algunos datos útiles, como los tipos de los personajes y la url a la api.

```python
#Search desired character
def buscarPersonaje(self, id : int) -> list:
    #Create url
    try:
        url = self.url+str(id)
    except TypeError:
        print("Id debe ser tipo int")

    querystring = {"":""}

    headers = {
        "x-inertia": "true",
        "x-inertia-version": "aa6e2214730f2161a1f77c9fa9074052"
    }

    response = requests.request("GET", url, headers=headers, params=querystring)

    #If character doesnt exist throw error
    if(response.status_code!=200):
        raise TypeError("Character doesnt exist or wrong code")

    #Add character name to list
    nombre = response.json()["props"]["card"]["name"]
    subnombre = response.json()["props"]["card"]["leader_skill_set"]["name"]
    tipo = response.json()["props"]["card"]["element"]
    awakeningType = response.json()["props"]["card"]["awakening_element_type"]
    nombre=nombre.replace('\n', '')
    dict = [subnombre+" "+nombre+" "+str(self.awakening[int(awakeningType)-1])+" "+str(self.type[int(tipo)])+" ("+str(id)+")"]
    dict.append(response.json()["props"]["card"]["link_skill_ids"])
    #Return list [Name, [Links skills]]
    return dict
```

La clase contiene un método para buscar el personaje el cual toma un id en forma de int y devuelve una lista. En este método se hace request a la pagina usando el id y esta nos devuelve un json. Tomamos ese json y creamos un diccionario con los valores que queremos. En este caso serian nombre, tipo y las habilidades para ver luego en que orden colocarlos.

La siguiente clase seria teamOptimizer:

```python
class teamOptimizer:
    #Create class with starting team
    def __init__(self, team):
        self.team = team
        self.links = self.checkLinks(team)
        self.linksMedia = self.checkLinksMedia(team)
```

El cual comienza con un constructor al que le pasamos el equipo.

```python
#Checks best team between all posible permutations
def optimizador(self):
    teamList = list(self.team)
    leader = teamList[0]
    friend = teamList[6]
    slice = teamList[1:6]
    #Create all possible permutations
    permutaciones = list(itertools.permutations(slice))
    #Check all permutations
    for i in range(len(permutaciones)):
        rotacion = []
        rotacion+= [leader]
        rotacion+= permutaciones[i]
        rotacion+= [friend]
        links = self.checkLinks(rotacion)
        #A = new team, b = old team. Checks all links and sees who have better
        a, b = 0,0
        for i in range(len(links)):
            if(links[i]>self.links[i]):
                a+=1
            elif(self.links[i]>links[i]):
                b+=1
        #If new team is better exchange them with base team
        if a > b :
            self.team = rotacion
            self.links = links

    return self.team
```

Tras esto tenemos el método que optimiza el equipo. Pasamos el equipo a lista y nos quedamos con el líder y amigo, estos personajes siempre tienen que ir al principio y al final del equipo. Con el resto, hacemos todas las permutaciones posibles e iteramos sobre estas. Si en algún caso el equipo es mejor que el actual, lo sustituimos.

```python
#Check all links of a team in a rotation and return a list with the number of links
def checkLinks(self, team : list) -> list:
    personajesList = team
    teamLinks = (self.linkComparator(personajesList[0][1], personajesList[1][1], personajesList[4][1]))
    teamLinks+=(self.linkComparator(personajesList[2][1], personajesList[3][1], personajesList[5][1]))
    teamLinks+=(self.linkComparator(personajesList[0][1], personajesList[1][1], personajesList[6][1]))
    teamLinks+=(self.linkComparator(personajesList[2][1], personajesList[3][1], personajesList[4][1]))
    teamLinks+=(self.linkComparator(personajesList[0][1], personajesList[1][1], personajesList[5][1]))
    teamLinks+=(self.linkComparator(personajesList[2][1], personajesList[3][1], personajesList[6][1]))
    return teamLinks
```

Ahora tenemos un método, al que introducimos un equipo y nos va devolviendo como de compatibles son los personajes.

```python
def linkComparator(self, links1 : list, links2:list, links3:list) -> list:
    links = []
    a = 0
    if len(links1)>len(links2):
        for i in range(len(links1)):
            for j in range(len(links2)):
                if links1[i] == links2[j]:
                    a+=1
    else:
        for i in range(len(links2)):
            for j in range(len(links1)):
                if links1[j] == links2[i]:
                    a+=1
    links.append(a)
    a = 0
    if len(links2)>len(links3):
        for i in range(len(links2)):
            for j in range(len(links3)):
                if links2[i] == links3[j]:
                    a+=1
    else:
        for i in range(len(links3)):
            for j in range(len(links2)):
                if links2[j] == links3[i]:
                    a+=1
    links.append(a)
    return links
```

Por último, en este método le pasamos los 3 personajes que queremos y nos devuelve una lista con como de compatibles son el primero con el segundo y el segundo con el tercero.

Ahora vamos con FastApi. En este he usado un solo archivo llamado main en el cual encontramos:

```python
app = FastAPI()
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")
```

Con eso hacemos que se monten las carpetas que vamos a usar y se inicie la app en FastApi

```python
@app.get("/")
async def root():
    response = RedirectResponse(url='./home')
    return response
```

Como en cualquier tipo de web app podemos definir si la llamada a esa url es de tipo post o get y cual será la url. En este caso será la url base. En esta creamos la respuesta para redirigir al usuario a home, que será nuestra pagina de inicio. Con RedirectResponse podemos crearlo, simplemente creamos un objeto de la clase indicando la url. Esto es parte de las cosas que implementa FastAPI.

```python
@app.get("/home", response_class=HTTPResponse)
async def home(request : Request):
    return templates.TemplateResponse("index.html", {"request":request})
```

Cuando llegamos a home, se lanzaría esta función get. En la cual, en la cabecera definimos el tipo de respuesta, que en este caso será HTTPResponse.

De esta manera podemos dar una repuesta de tipo HTTP, pero hay varias. Por defecto, tenemos JSONResponse, la cual devuelve un JSON. También existe HTML response, la cual nos permite crear en el return una pagina web en forma de string y devolverla como una pagina valida. Además, podemos sobrescribir el tipo de respuesta dentro de las funciones, así no es solo el tipo que definamos desde un inicio, puede cambiar dependiendo de varios factores.

A la función le pasamos como parámetro Request, que es un objeto de FastAPI que contiene información varia sobre la request, lo mismo que en otros tantos frameworks del estilo.

Y por último devolvemos un template pasándole el nombre del archivo que estará dentro de la carpeta templates y la request. Para hacer esto FastApi usa Jinja, que es un motor de plantillas web para Python, es decir, usando esa tecnología podemos crear paginas HTML en Python.

La pagina que devolvemos se vería así:

```html
</head>
<body>
    <h1>DokkanBestOrder</h1>
    <h4>Busca los codigos de personajes en <a href="https://dokkan.fyi/characters">https://dokkan.fyi/characters</a></h4>
    <form action="./team" method="post" enctype="aplication/x-www-form-urlencoded">
        <label for="leader">Lider:</label>
        <input type="number" id="leader" name="leader" value="0" required ><br><br>
        <label for="char2">Segundo personaje:</label>
        <input type="number" id="char2" name="char2" value="0" required><br><br>
        <label for="char3">Tercer personaje:</label>
        <input type="number" id="char3" name="char3" value="0" required><br><br>
        <label for="char4">Cuarto personaje:</label>
        <input type="number" id="char4" name="char4" value="0" required><br><br>
        <label for="char5">Quinto personaje:</label>
        <input type="number" id="char5" name="char5" value="0" required><br><br>
        <label for="char6">Sexto personaje:</label>
        <input type="number" id="char6" name="char6" value="0" required><br><br>
        <label for="friend">Amigo:</label>
        <input type="number" id="friend" name="friend" value="0"><br><br>
        <input type="submit" id="submit" value="Submit">
    </form>
    <p>(Puede tardar unos segundos tras pulsar submit)</p>
    <a href="https://github.com/Maquinero123456/dokkanBestOrderWeb">GitHub</a>
</body>
</html>
```

Como cualquier pagina web normal, pero podemos introducir código de Python como haríamos con java en las practicas del bloque 2 y 3. En este caso yo no he usado pero se haría poniendo entre {% %} para bloques de código como if o for y {{ }} para variables.

En este caso, al pulsar submit, enviamos el formulario a la url “./team” que la tenemos definidas y el tipo de encode que vamos a usar será “aplication/x-www-form-urlencoded” para que podamos conseguir los valores del formulario de manera sencilla de la siguiente forma:

```python
@app.post("/team", response_class=HTMLResponse)
async def team(request : Request, leader : int = Form(), char2 : int = Form(), char3 : int = Form(), char4: int = Form(), char5 : int = Form()
    if(friend==0):
        friend=leader

    personajes = []
    search = characterSearch()

    personajes.append(search.buscarPersonaje(leader))
    personajes.append(search.buscarPersonaje(char2))
    personajes.append(search.buscarPersonaje(char3))
    personajes.append(search.buscarPersonaje(char4))
    personajes.append(search.buscarPersonaje(char5))
    personajes.append(search.buscarPersonaje(char6))
    personajes.append(search.buscarPersonaje(friend))

    optimizar = teamOptimizer(personajes)
    equipo = optimizar.optimizador()

    return templates.TemplateResponse("team.html", {"request": request, "id1": personajes[0][0], "id2": personajes[1][0], "id3": personajes[2]
```

Como se puede ver, ahora definimos la función como post, ya que enviamos los datos del formulario. Y en la función tenemos muchos parámetros, esto es debido a que, una forma sencilla de obtener los datos de un form en FastApi es poniendo como parámetros variables con el nombre de los input del formulario que queremos obtener. Si nos fijamos, todas tienen el nombre de un input del form.

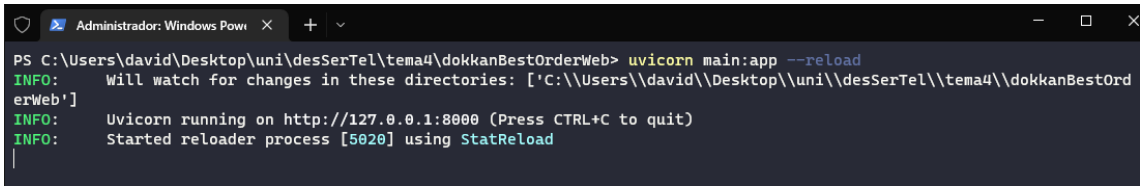
Dentro de la función, primero compruebo que el amigo haya un valor distinto a 0, en caso contrario lo sustituyo por el del líder. Ahora hacemos la búsqueda de todos los personajes y optimizamos el equipo.

Tras hacer esto devolvemos un template, en este caso el de team y como parámetros, ahora no solo devolvemos la request, sino que también el nombre de cada uno de los personajes para poder usarlos en el template de la siguiente manera:

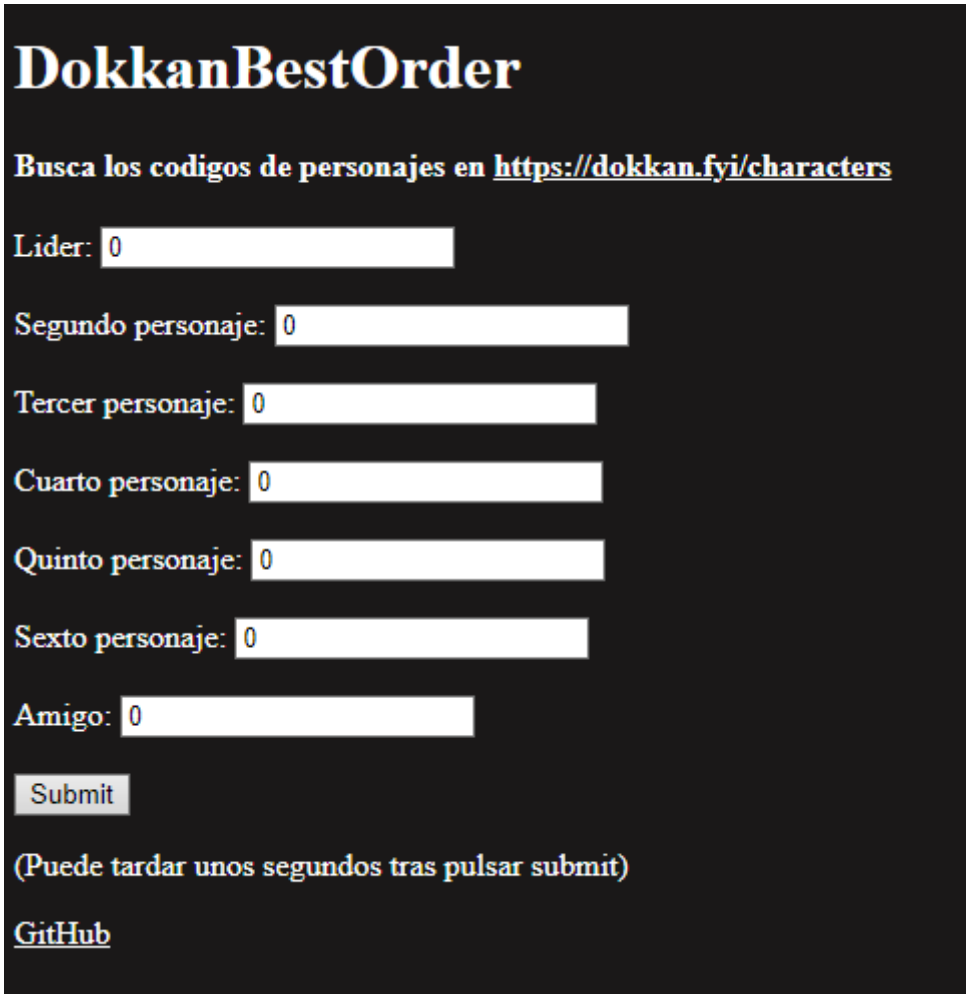
```html
<body>
    <h1>Equipo introducido:</h1>
    <ol>
        <li>{{id1}}</li>
        <li>{{id2}}</li>
        <li>{{id3}}</li>
        <li>{{id4}}</li>
        <li>{{id5}}</li>
        <li>{{id6}}</li>
        <li>{{id7}}</li>
    </ol>
    <hr>
    <h1>Equipo ordenado:</h1>
    <ol>
        <li>{{id8}}</li>
        <li>{{id9}}</li>
        <li>{{id10}}</li>
        <li>{{id11}}</li>
        <li>{{id12}}</li>
        <li>{{id13}}</li>
        <li>{{id14}}</li>
    </ol>
```

Cada uno de los nombres de las variables, son de uno de los parámetros que hemos colocado en el return anterior, así, de esta manera podemos variar la template dependiendo del resultado que nos de la aplicación.

Para probarlo, simplemente tenemos que abrir una terminal e irnos a la carpeta de nuestra app:



Simplemente nos tenemos que ir la dirección que nos dice. Conforme vayamos actualizando cosas en la aplicación, se ira actualizando sola.



Eso es lo que vemos si introducimos la url en el navegador y si introducimos unos datos validos:

Que seria el equipo ya optimizado.

Así que como vemos, FastAPI es un framework bastante útil para hacer algún proyecto rápido o si queremos hacer algo mas grande, es posible al ser bastante escalable y rápido. Además de usar Python, lo que nos permite escribir backend de manera rápida y simple, y usar jinja para las páginas e introducir código en ellas.