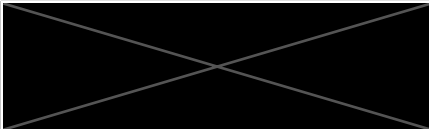


React - Part 2



AMONG THE STARS

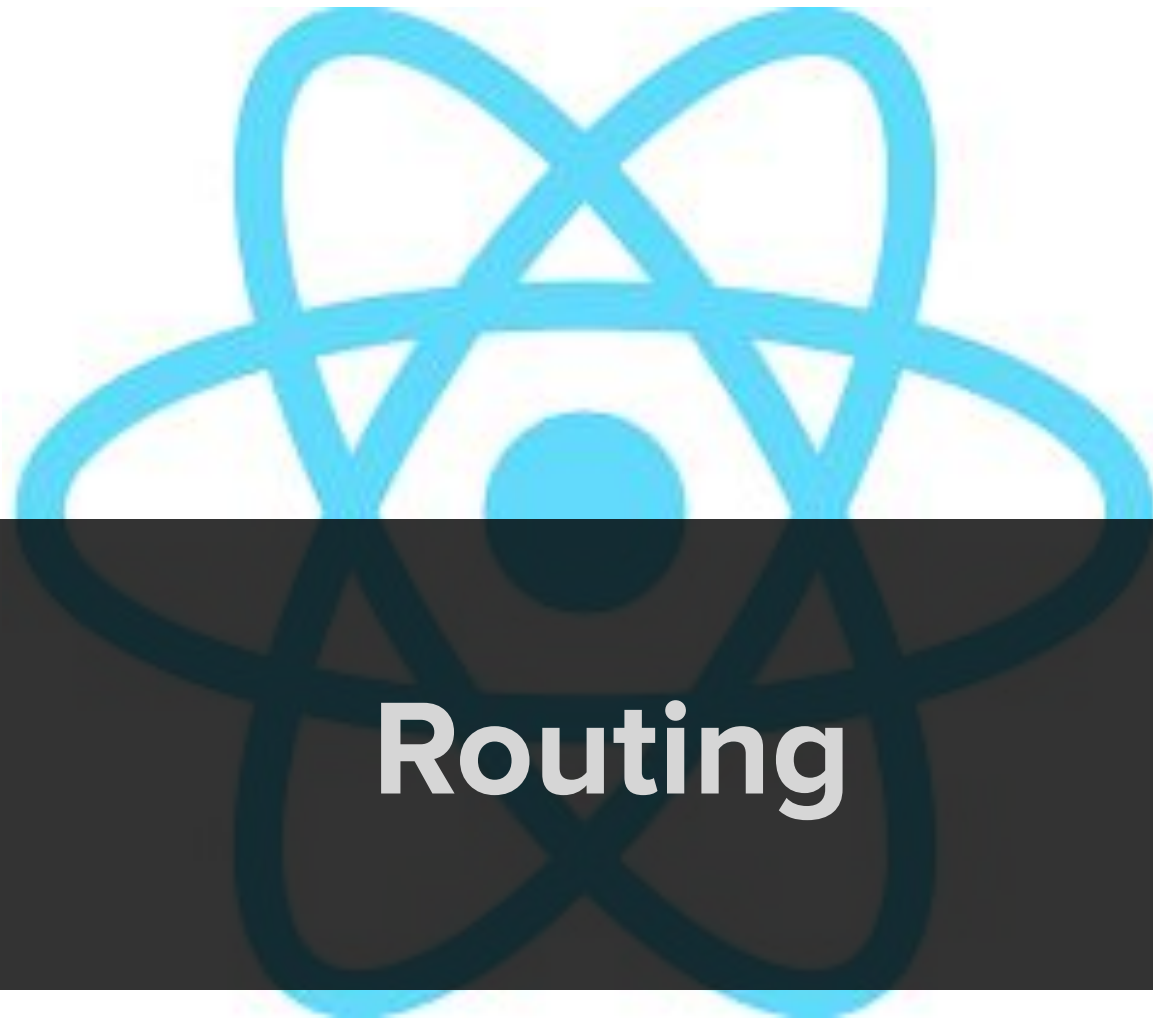
[Kies een challenge](#)

[▶ Bekijk de trailer](#)

Contents

1. Routing
2. useEffect
3. Backend communication
4. Component frameworks

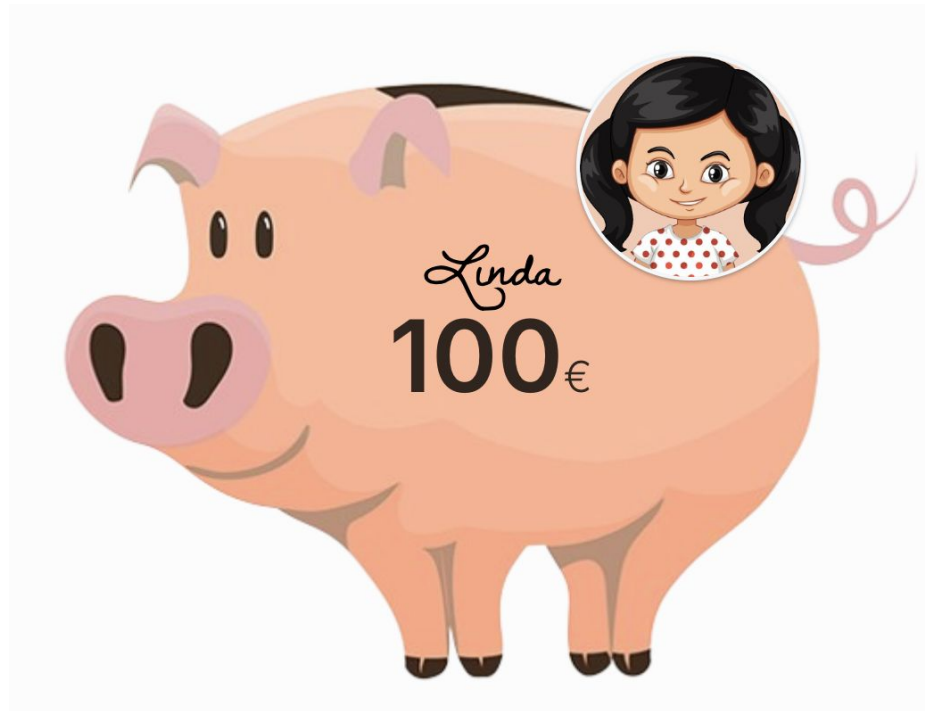




Routing

= **Attaching a component to a URL**

`localhost:5173/piggybanks/1`



Navigating from one route (URL) to another does not trigger a full page reload:
client side routing

A router library is not included in React, you can choose your own routing solution, but [React Router](#) is the de facto standard

Configuration is usually done at the top level (App.tsx)

```
function App() {  
  return (  
    <BrowserRouter>  
      <Routes>  
        <Route path="/piggybanks/:id" element={<PiggyBankDetail/>}/>  
        <Route path="/piggybanks" element={<PiggyBankList/>}/>  
        <Route path="/" element={<Navigate to="/piggybanks"/>}/>  
      </Routes>  
    </BrowserRouter>  
  )  
}
```

`<BrowserRouter>` activates routing using the [browser history API](#).
There are also routers for React native, testing,...

`<Routes>` will select the `<Route>` that best matches the URL.

(`/` matches any URL, `<Navigate>` redirects to another route)

```
<Route path="/piggybanks/:id">
```

:<param-name> can be used to pass a parameter
The React framework recognizes this pattern and passes the id from the url to the component using the **useParams hook**.

```
function Piggybank() {  
  const { id } = useParams()  
  ...  
}
```

User navigates to /piggybanks/3 => useParams returns 3

You can use two ways to navigate to a route.

<Link to="url">

Used to put navigation in JSX. Generates <a> tag behind the scenes

```
<Link to="/about">About</Link>
```

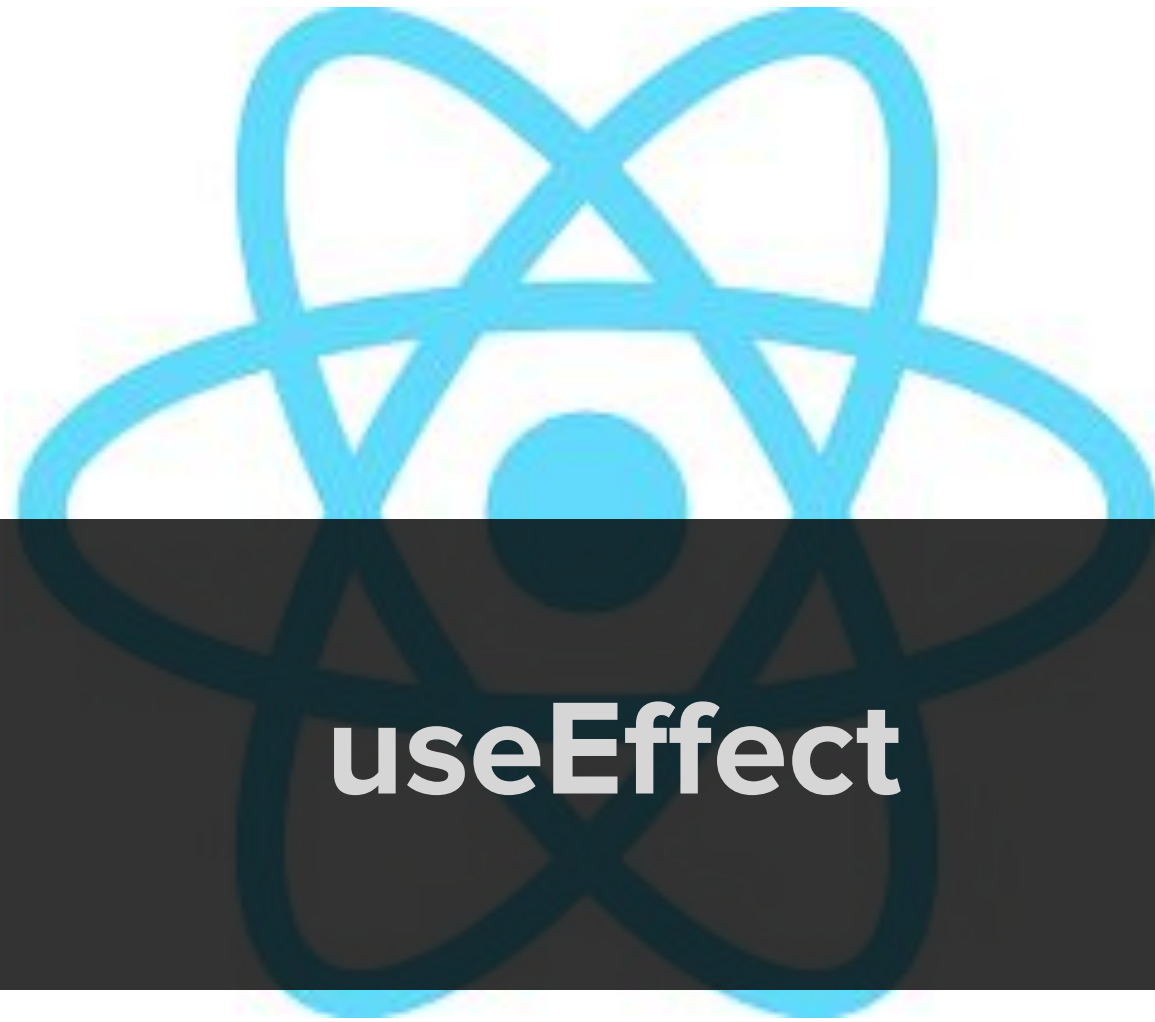
useNavigate hook

Used to handle dynamic navigation in Javascript

```
import {useNavigate, Link} from "react-router-dom";

function SomeComponent() {
  const navigate = useNavigate();
  const createRecipe = async (data) => {
    const newRecipe = await storeRecipeInDB(data);
    navigate(`/recipes/${newRecipe.id}`);
  }

  return (
    <Link to="/home"><button>Go back</button></Link>
  );
}
```

Recap

UI = **f**(**state**)



Recap

incoming state (props)

balance
owner

callbacks (props)

onClick,...



private state

showFields



useState hook



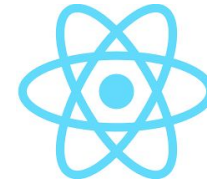
PiggyBank component

`setShowField(true)`
"I have a state
change!"

"Thanks for calling
me bro, here is the
JSX for my new
state"

"Hope Badge likes
the props I passed
on ;-)"

React



"Roger that, I will **re-call**
you and `showFields` will
be true when I do so"
`PiggyBank()`

"Thx! I will check if it
differs from what is on the
screen and repaint if
necessary"

"Oh: your JSX contains a
component `<Badge>`, I will
call `Badge()` first"

state change

in MyComponent (or parent of MyComponent)



render = call
MyComponent()



paint
(only if necessary)
returned JSX => VDOM => DOM



after paint
call **useEffect** hooks

Side effects

Functional component = **PURE function**

Called again with the same props & internal state it should return the same JSX to be rendered

But “sometimes” an application has to **do something else** then render data = **apply side effects**

If caused by and **interaction** (click, swipe,...) this is done in **event handlers**

```
<OwnerBadge onClick={() => setShowfields(true)}/>
```

But what if you want to do something that can not be placed in an event handler?

useEffect

useEffect allows you to **synchronize an external API** with Reacts declarative/reactive way of doing things

Examples

- get data from a REST-endpoint
- (un)subscribe to/from a chat room web socket
- write/read a setting to/from local storage
- use a native browser API that needs cleanup (`setInterval`,...)
- play a sound when a component property changes
- ...

[LEARN REACT](#) > [ESCAPE HATCHES](#) >

Synchronizing with Effects

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. *Effects* let you run some code after rendering so that you can synchronize your component with some system outside of React.

```
import { useEffect } from 'react';
```

```
useEffect(() => {  
  /* function that is executed when the effect runs */  
  return () => { /* clean up function, optional */}  
},  
  [/* dependency list: defines when we should run the effect again*/]  
);
```




```
import { useEffect } from 'react';
```

```
useEffect(() => {  
  /* function that is executed when the effect runs */  
  return () => { /* clean up function, optional */}  
},  
  [/* dependency list: defines when we should run the effect again*/]  
);
```

- Used to run “side effects” **after** a render.
- Function with two arguments:
 - a function that is executed when the useEffect hook runs
 - Can return a function that runs when the component ‘unmounts’ (is removed from the screen) to clean up.
 - a “dependency list” (optional): an array of variables that trigger a re-run of the function when they change.
- Dependency list determines when to run the effect (again).
- Always runs at least once (after the first render = ‘mount’)

Rules of hooks

useState and *useEffect* are the most famous React hooks, but there are others like *useRef*, *useCallback*,... (we will discuss these later on), and can also write your own [custom hooks](#).

But wait! There are some rules ...

Hooks can only be used if we adhere to [“the rules of hooks”](#)

1. Name always start with ‘use’
2. Only call hooks on the top-level (not conditionally, not in a loop,...)!
3. Only call hooks from React functions
 - a. React components (are functions!)
 - b. Custom hooks

Why?

Because React relies on the order in which hooks are called to return the right value



The React logo, a stylized blue atom with three intersecting elliptical orbits and a central blue circle, is positioned in the background. A dark gray horizontal bar spans the width of the slide, partially obscuring the logo and serving as a background for the title text.

Backend communication

React is a **library** rather than a **framework**... it doesn't give you a turnkey for things like *networking, routing, forms,...*



flexibility



making choices
extra configuration



To handle backend HTTP calls, you can use the standard **fetch** API or pick a more advanced solution like **axios**, **xior**, **ky**...

On top of that you will need **useEffect** to initiate the call. Typically you want to do this after 'first render' of the component (= after the first time `MyComponent()` is called).

Pass an empty dependency array to accomplish this.

```
export function Todos() {
  const [todos, setTodos] = useState<Todo[]>([]);

  useEffect(() => {
    async function fetchTodos() {
      const {data} = await axios.get<Todo[]>(URL);
      setTodos(data);
    }
    fetchTodos();
  }, []);

  return (
    <div>
      <ul>{todos.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>
    </div>
  )
}
```

useEffect does not accept an async function, we have to work with a separate async function that we call immediately

Empty dependency array: the effect runs only once, on first render

Asynchronous communication **takes time** and **can go wrong**...so we need to provide **loading** and **error** handling...this leads to a lot of boilerplate code

```
export function Todos() {
  const [todos, setTodos] = useState<Todo[]>([]);
  const [isLoading, setIsLoading] = useState(true);
  const [isError, setIsError] = useState(false);

  useEffect(() => {
    async function fetchTodos() {
      try {
        const response = await axios.get<Todo[]>(URL);
        setTodos(response.data);
      } catch (e) {
        setIsError(true);
      }
      setIsLoading(false);
    }
    fetchTodos();
  }, []);

  if (isLoading) {
    return <div>Loading todos</div>
  }

  if (isError) {
    return <div>Todos could not be loaded</div>
  }

  return (
    <div>
      <ul>{todos.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>
    </div>
  )
}
```

React Query

[React Query](#) is a library that reduces boilerplate code by handles data fetching (and other things like caching) for you and. It does this by **using useEffect** and **useState** under the hood.

TanStack Query v5

Powerful asynchronous state management for TS/JS, React, Solid, Vue, Svelte and Angular

There are (of course) alternatives...

Comparison | React Query vs SWR vs Apollo vs RTK Query vs React Router

React Query

Use *QueryClientProvider* to supply a *QueryClient* instance to your code.

(This is a “context”, more on that later!)

```
const queryClient = new QueryClient()

function App() {
  return (
    // Provide the client to your App
    <QueryClientProvider client={queryClient}>
      <Todos />
    </QueryClientProvider>
  )
}
```

React Query

Define functions that do the actual data manipulation.
This can be *anything that returns a promise*.

```
export async function getTodos(){
  const {data: todos} = await axios.get<Todo[]>('/todos');
  return todos;
};

export function postTodo(todo: Todo): Promise<Todo> {
  return axios.post('/todos', todo)
};
```

React Query

Uses these functions in custom hooks to get (*useQuery*) and update (*useMutation*) data

```
function Todos() {  
  const { isLoading, isError, data: todos } =  
    useQuery({ queryKey: ['todos'], queryFn: getTodos })  
  
  if (isLoading) {  
    return <div>Loading todos</div>  
  }  
  
  if (isError) {  
    return <div>Todos could not be loaded</div>  
  }  
  
  return (  
    <div>  
      <ul>{todos.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>  
    </div>  
  )  
}
```

React Query

```
function Todos() {
  const queryClient = useQueryClient()
  const query = useQuery({ queryKey: ['todos'], queryFn: getTodos })

  const mutation = useMutation({
    mutationFn: postTodo,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['todos'] })
    },
  })

  return (
    <div>
      <ul>{query.data?.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>

      <button
        onClick={() => {
          mutation.mutate({
            id: Date.now(),
            title: 'Do Laundry',
          })
        }}
      >
        Add Todo
      </button>
    </div>
  )
}
```

(error and loading flags omitted for clarity)

React Query

The **query 'key'** [`'todos'`] is used by React Query internally to cache the data and refetch when

- the browser window/tab is selected
- `invalidateQueries({ queryKey: ['todos'] })` is called

`useQuery` immediately returns the cached data and does a refetch in the background

```
const query = useQuery({ queryKey: [ 'todos' ], queryFn: getTodos })

const mutation = useMutation({
  mutationFn: postTodo,
  onSuccess: () => {
    // Invalidate and refetch
    queryClient.invalidateQueries({ queryKey: [ 'todos' ] })
  },
})
```



React Query (or similar libraries) reduce the amount of ‘boilerplate’ code significantly.

Check the **albums** demo on Canvas, compare with and without RQuery

```
const URL = 'https://jsonplaceholder.typicode.com/albums';

export function Albums () {
  const [albums, setAlbums] = useState( initialState: []);
  const [isLoading, setIsLoading] = useState( initialState: true);
  const [isError, setIsError] = useState( initialState: false);

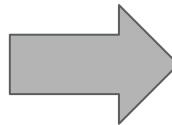
  useEffect( effect: () => {
    async function fetchAlbums() {
      try {
        const response = await axios.get(URL);
        setAlbums(response.data);
        setIsLoading( value: false);
      } catch (e) {
        setIsError( value: true);
        setIsLoading( value: false);
      }
    }

    fetchAlbums();
  }, deps: []);

  if (isLoading) {
    return <div>Error loading albums</div>
  }

  if (isError) {
    return <div>Albums could not be loaded</div>
  }

  return (
    <div className="albums">
      {albums.map(({id, title}) => (
        <h1 key={id}>{title}</h1>))}
    </div>
  )
}
```



```
const URL = 'https://jsonplaceholder.typicode.com/albums';

async function getAlbums() : Promise<Album[]> { Show usages new *
  const {data: albums} = await axios.get<Album[]>(URL);
  return albums;
}

export function Albums() : Element { Show usages new *
  const {isLoading, isError, data: albums} =
    useQuery(
      {
        queryKey: ['albums'],
        queryFn: getAlbums
      }
    );

  if (isLoading) {
    return <div>Loading albums</div>
  }

  if (isError || typeof albums == "undefined") {
    return <div>Albums could not be loaded</div>
  }

  return (
    <div className="albums">
      {albums.map(({id, title} : Album) : Element => (
        <h1 key={id}>{title}</h1>))}
    </div>
  )
}
```

Lots of features...

Query Functions

Network Mode

Parallel Queries

Dependent Queries

Background Fetching Indicators

Window Focus Refetching

Disabling/Pausing Queries

Query Retries

Paginated Queries

Infinite Queries

Placeholder Query Data

Initial Query Data

Prefetching

Mutations

Query Invalidation

Invalidation from Mutations

Updates from Mutation Responses

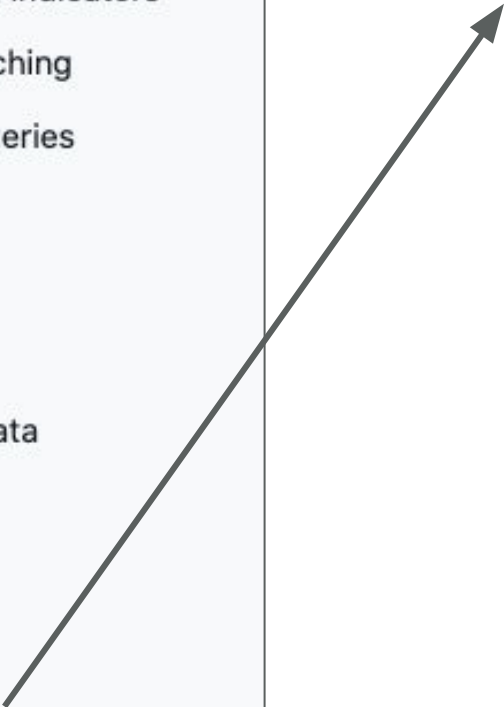
React query will (out of the box) retry 4 times (can be configured) to load data when a 404 is encountered

Lots of features...

Query Functions

- Network Mode
- Parallel Queries
- Dependent Queries
- Background Fetching Indicators
- Window Focus Refetching
- Disabling/Pausing Queries
- Query Retries
- Paginated Queries
- Infinite Queries
- Placeholder Query Data
- Initial Query Data
- Prefetching
- Mutations
- Query Invalidation
- Invalidation from Mutations
- Updates from Mutation Responses

React query can automatically refetch data after a mutation (http post,...)



Lots of features...

Query Functions

Network Mode

Parallel Queries

Dependent Queries

Background Fetching Indicators

Window Focus Refetching

Disabling/Pausing Queries

Query Retries

Paginated Queries

Infinite Queries

Placeholder Query Data

Initial Query Data

Prefetching

Mutations

Query Invalidation

Invalidation from Mutations

Updates from Mutation Responses

You can specify that a query only runs when a specific condition is met (eg. that another query is finished)

```
// Get the user
const { data: user } = useQuery({
  queryKey: ['user', email],
  queryFn: getUserByEmail,
})

const userId = user?.id

// Then get the user's projects
const {
  status,
  fetchStatus,
  data: projects,
} = useQuery({
  queryKey: ['projects', userId],
  queryFn: getProjectsByUser,
  // The query will not execute until the userId exists
  enabled: !!userId,
})
```


Lots of features...

Query Functions

Network Mode

Parallel Queries

Dependent Queries

Background Fetching Indicators

Window Focus Refetching

Disabling/Pausing Queries

Query Retries

Paginated Queries

Infinite Queries

Placeholder Query Data

Initial Query Data

Prefetching

Mutations

Query Invalidation

Invalidation from Mutations

Updates from Mutation Responses

React query can refetch data when the browser tab is refocused by the user

The React logo, a stylized blue atom with three intersecting elliptical orbits and a central blue circle, is positioned in the background. A dark gray horizontal band spans the width of the slide, partially obscuring the logo and serving as a background for the title text.

Component frameworks

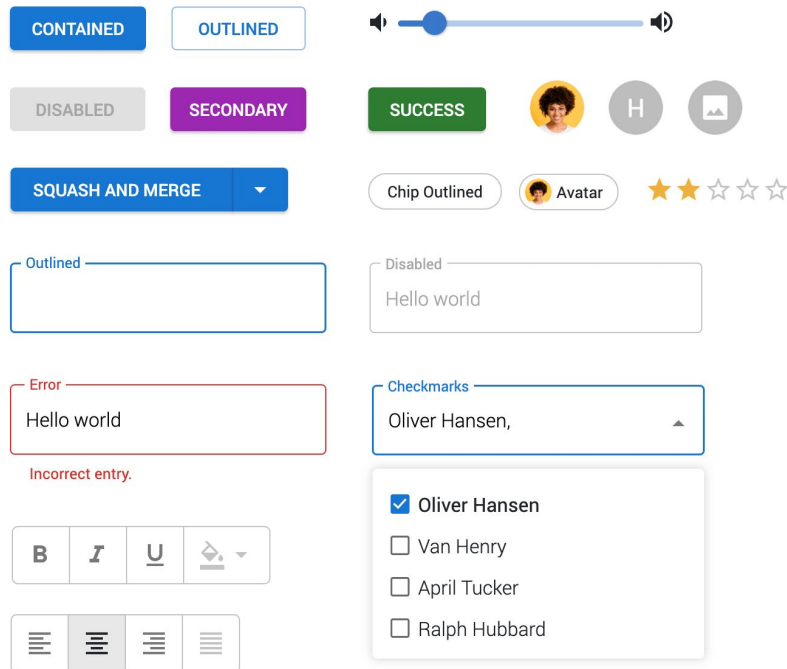
Almost every (large) application needs general functionality like lists, menus, cards, forms, app-bars, drawers,...



Do not reinvent the wheel

Component framework

A **collection of components** that follow the rules of a certain **design system**. Can be adapted to your needs using CSS and theming.



Components

INPUTS

Autocomplete

Button

Button Group

Checkbox

Floating Action Button

Radio Group

Rating

Select

Slider

Switch

Text Field

Transfer list

Toggle button

DATA DISPLAY

Avatar

Badge

Chip

Divider

Icons

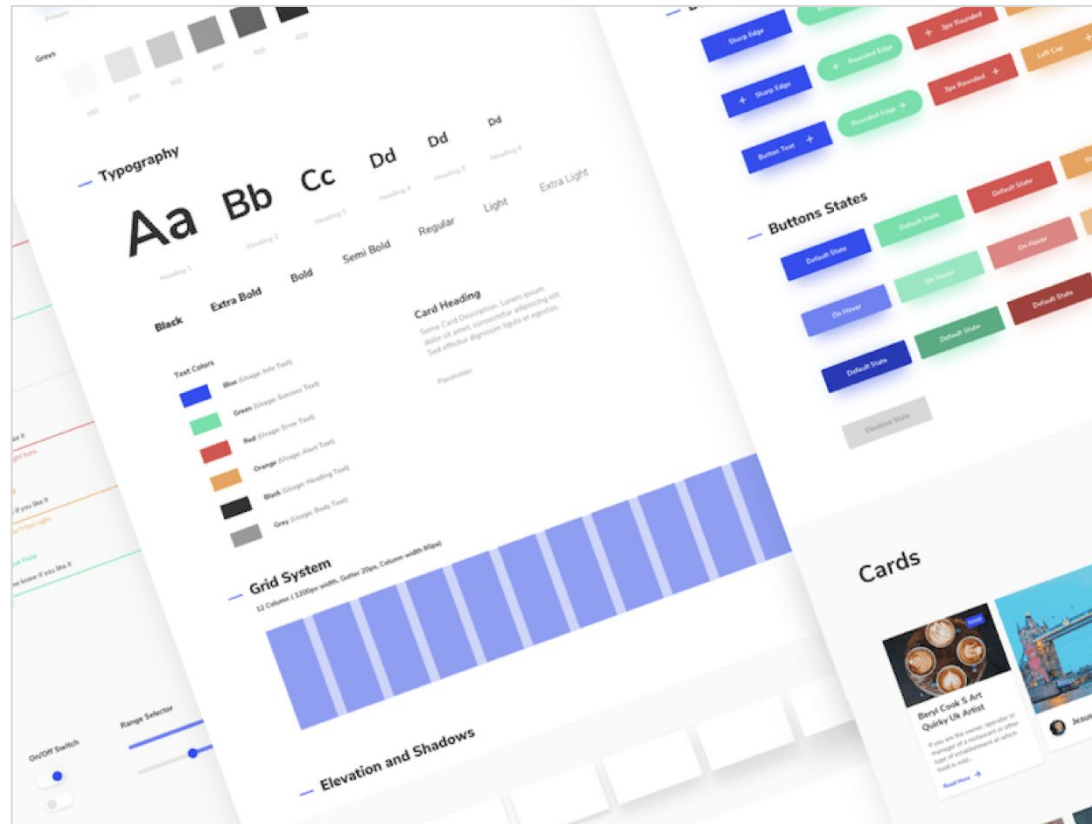
Material Icons

List

Table

Design system

A collection of **guidelines**, **design tokens** and **resources** (icons, css files, components, code snippets, etc.) that together form a **consistent set of rules** and a **toolbox for designing** a user interface.



Design tokens

The atomic elements that make up a design system
(*colors, shadows, fonts, spacing/grid, animations etc...*)

Color



color.input.background.error

Radius



radius.medium

Elevation



elevation.level.2

Ease



ease.appear.emphasize

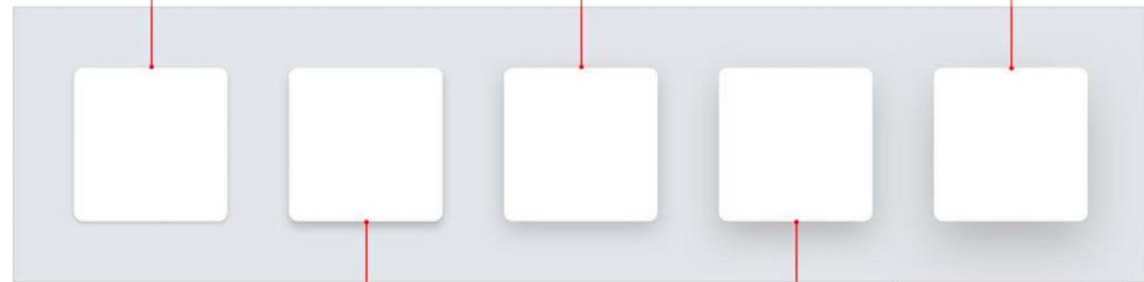


SO MANY OPTIONS, SO LITTLE TIME

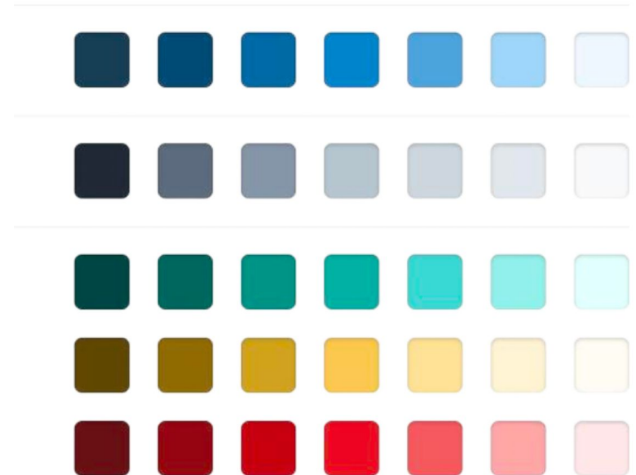
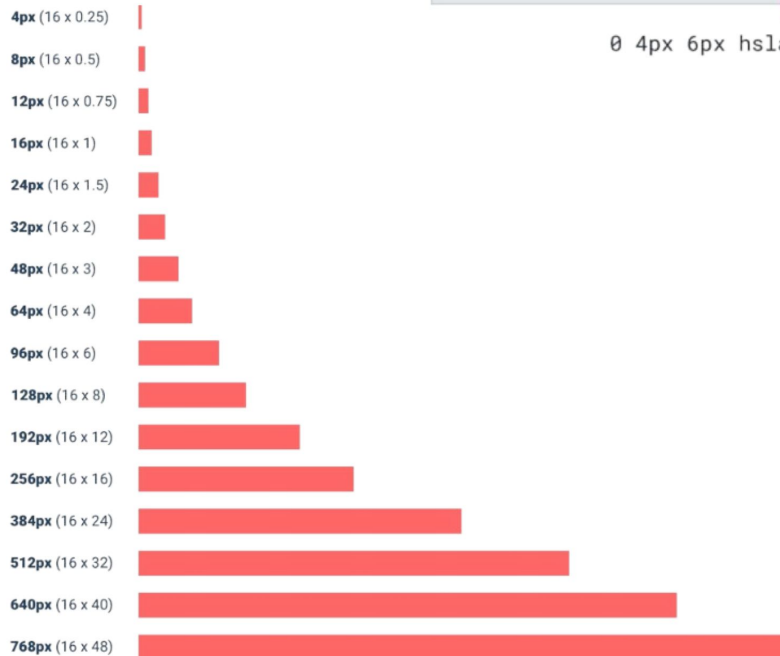


In a design system, choices are made for you to reduce the number of options and create **consistent interfaces**.

```
0 1px 3px hsla(0,0%,0%,.2); 0 5px 15px hsla(0,0%,0%,.2); 0 15px 35px hsla(0,0%,0%,.2);
```

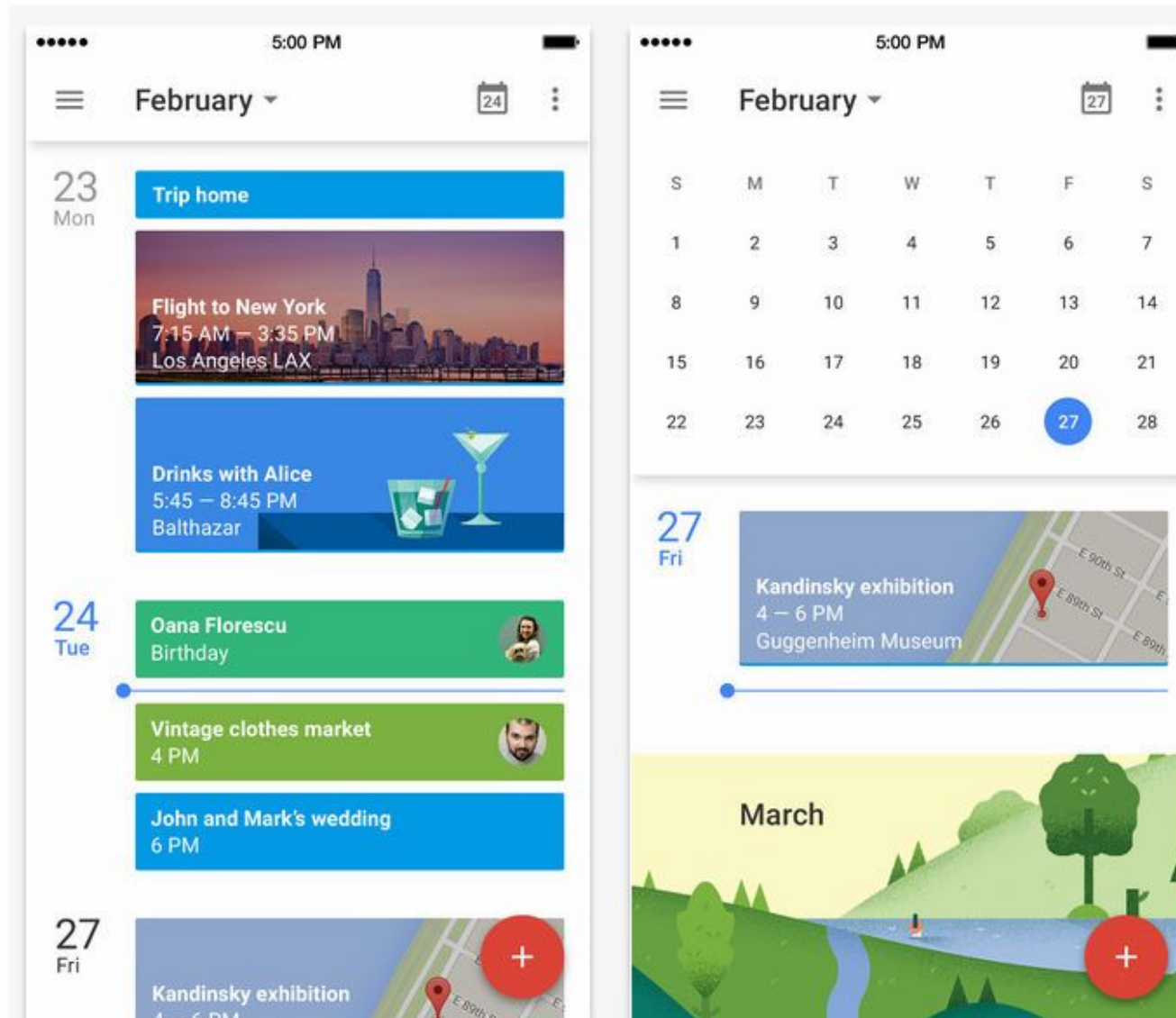


```
0 4px 6px hsla(0,0%,0%,.2); 0 10px 24px hsla(0,0%,0%,.2);
```



Material design

Example of a design system. It is used by Google in all its products.



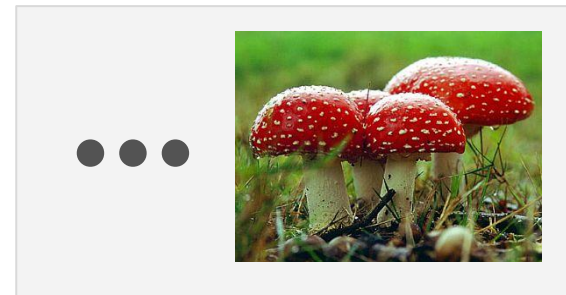
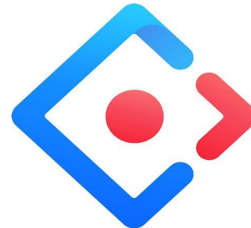
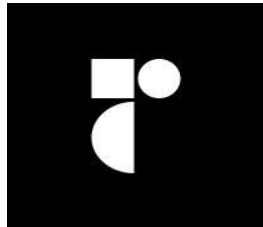
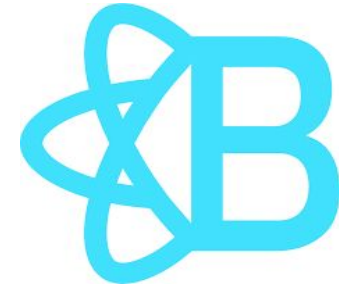
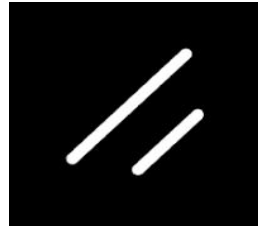
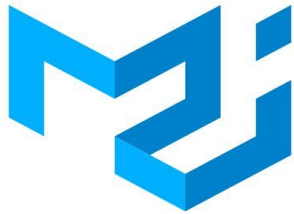
Material design



<https://material.io/design/>

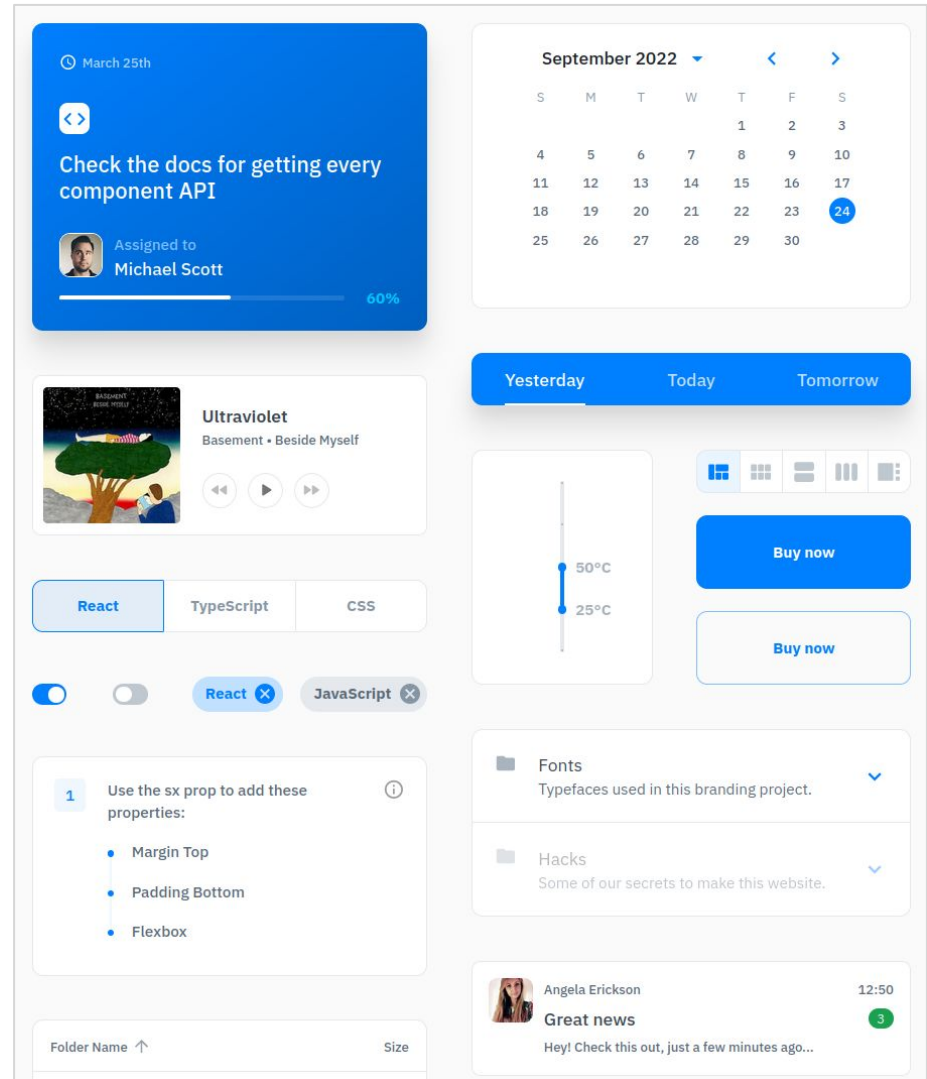
Component frameworks

There are **a lot** of component frameworks. Each framework follows a specific design system (unless it is [headless](#)).



MUI

One of the **most used and extensive** component frameworks. We will apply it in the piggybank exercise and in the example code, but **you are free to pick another!** MUI adheres to the **material design system**.



“Light” approach

Do not use a component framework, style your div’s with a CSS framework like bootstrap, [tailwind](#),...(or combine this with a [Headless](#) component framework)
Create your own components using these code ‘blocks’.

“Heavy” approach

Use a component framework like [MUI](#), [Mantine](#), [React-Bootstrap](#), [Ant Design](#), [PrimeReact](#), ...



Powerful components as black boxes with clearly defined props
Less wrapping in own components is needed

`<Button>Save</Button>` **VS.**

```
<div className="text-white bg-blue-700 hover:bg-blue-800 focus:ring-4 focus:ring-blue-300 font-medium rounded-lg text-sm px-5 py-2.5 mr-2 mb-2 dark:bg-blue-600 dark:hover:bg-blue-700 focus:outline-none dark:focus:ring-blue-800">Save</div>
```



A stronger vendor lock-in
Less flexible (adaptable, reusable) than e.g. tailwind code blocks

Most of the component frameworks promote a CSS-in-JS approach which means you can **embed styles in Javascript**.

This is not necessary a bad thing. Remember we talked about JSX being HTML-in-JS and **'keep together what changes together'**.

Moreover, it gives you a lot of extra power like dynamic styles, dynamic theming and dynamic responsiveness (*with dynamic we mean it is just JS so you can use if tests, ternary expressions etc to calculate styles directly from your state*)

For example MUI uses [emotion](#) as its styling engine. Every component has an sx property that accepts a JS object with CSS styles.

```
<CircularProgress  
  sx={{ display: "block", mt: aJSVariable, mx: someTSFunction() }}  
/>
```

(mt and mx are MUI shorthands for margin settings)

This does not mean you cannot change global styles for all your components (like colors, fonts,...)! This is usually done through a [theming](#) mechanism.

That's all Folks!

