6.104 · Software Design · MIT · Fall 2025

back-end implementation

Daniel Jackson

your goals for today's class

solidify an important idea from last lecture
why asynchronous requests are key to web apps

understand what goes on client vs server
essential to understanding why client side syncs aren't enough

remember list functionals
an essential programming paradigm

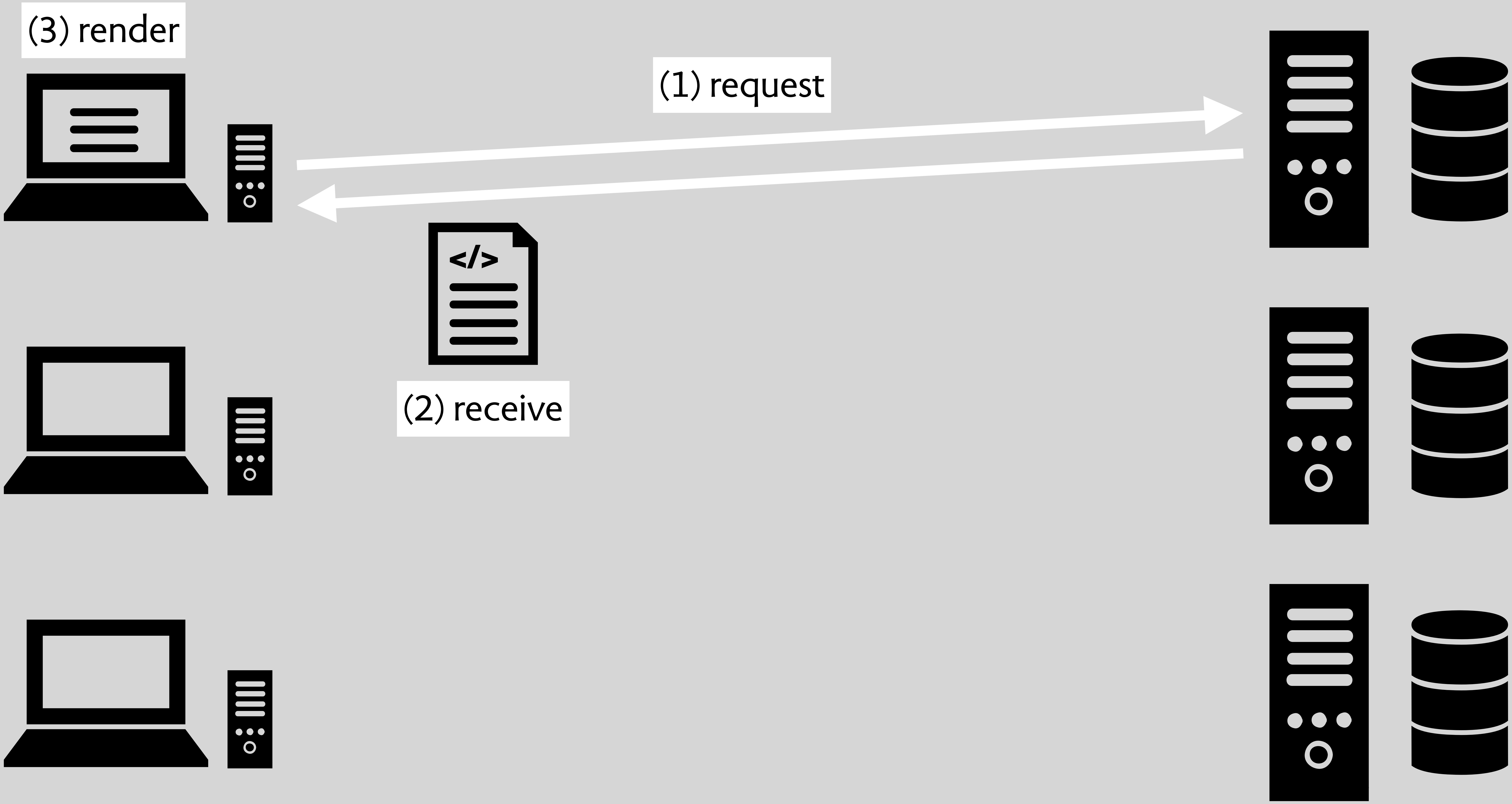
draw connections to relational & collection databases
idea of operating on lists

understand how where clause of sync works
a pipeline of functions on a list (of frames)

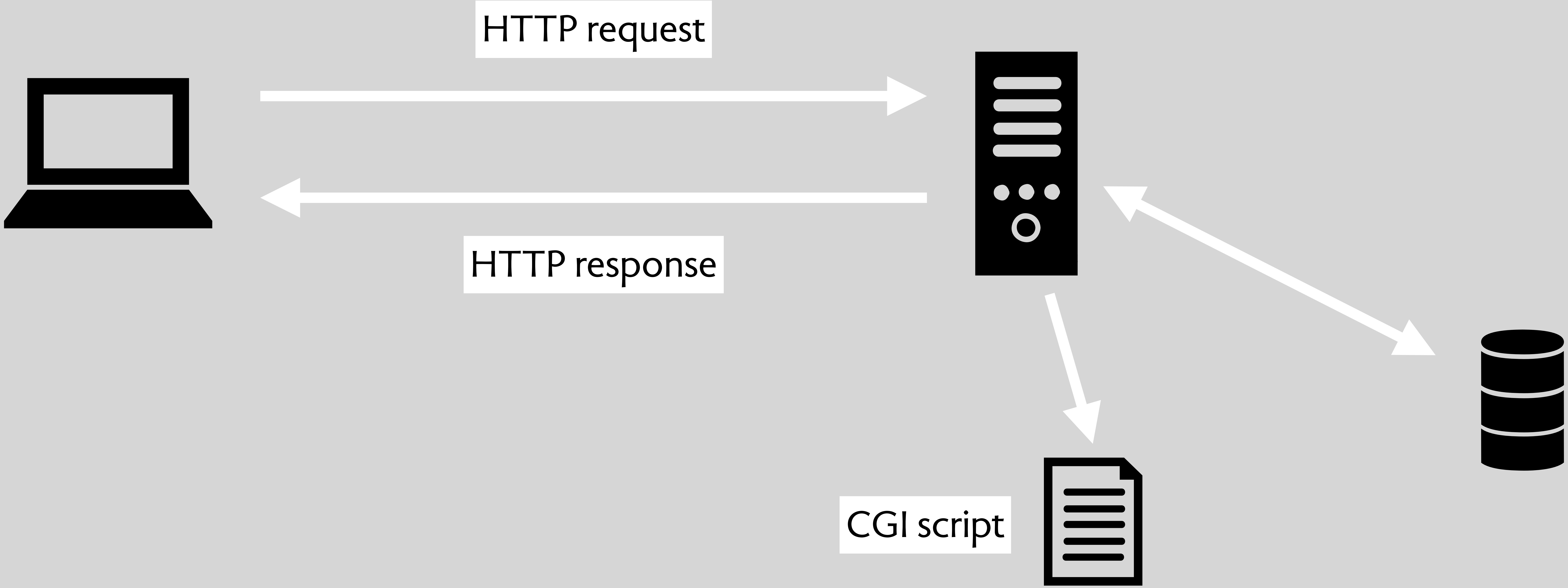
recap

why asynchronous requests matter

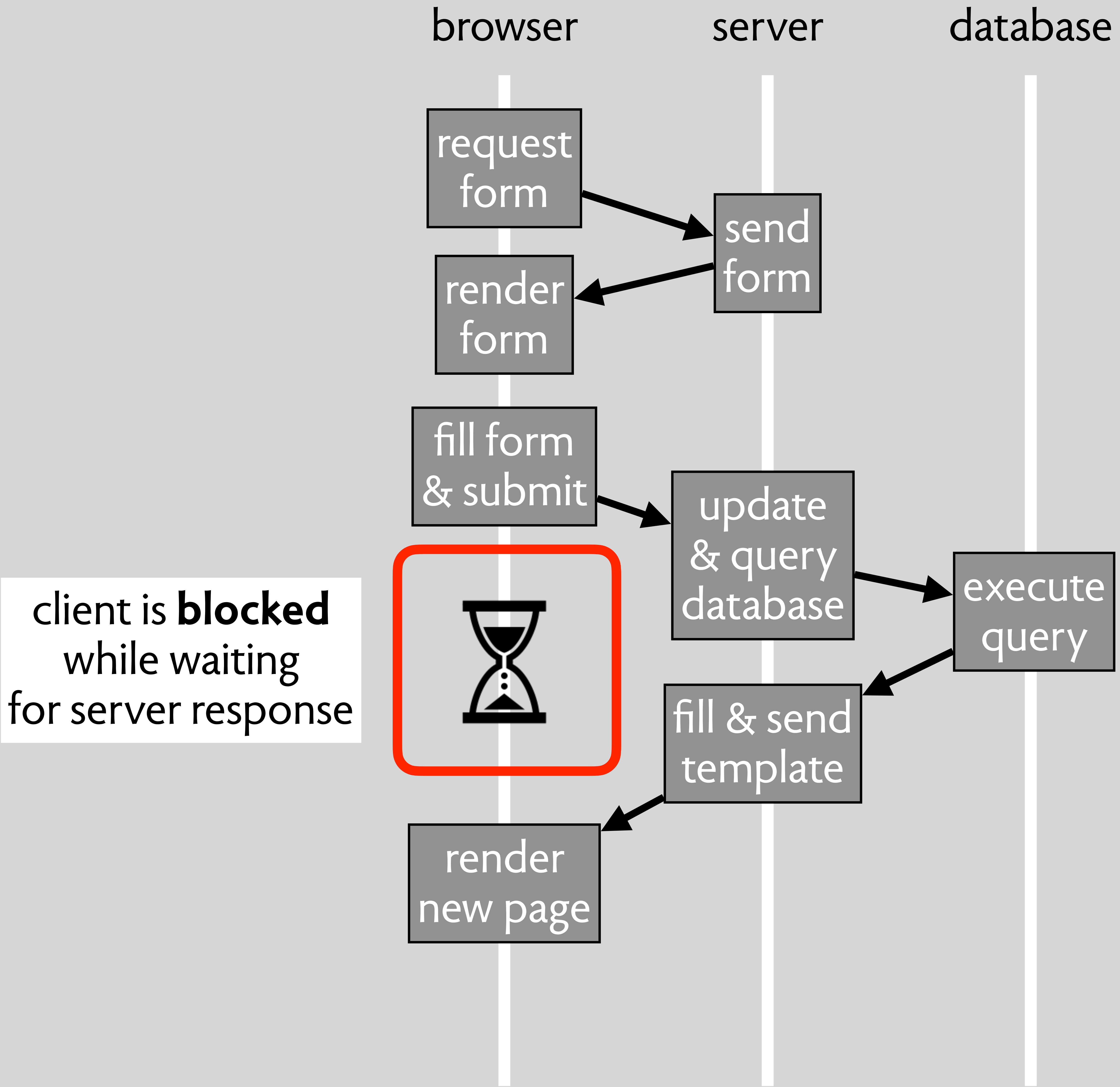
TBL’s web (1991)



common gateway interface (1993)

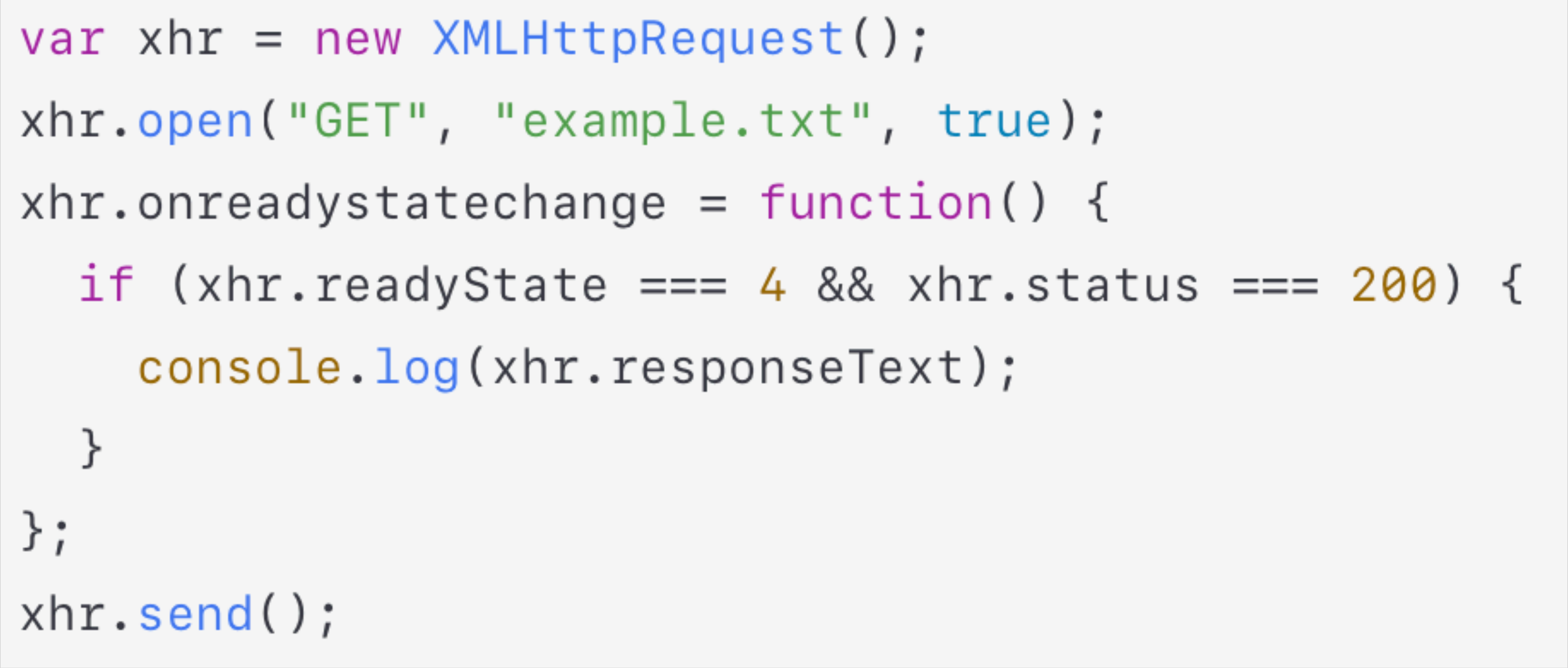


the flow for a “multi-page” app

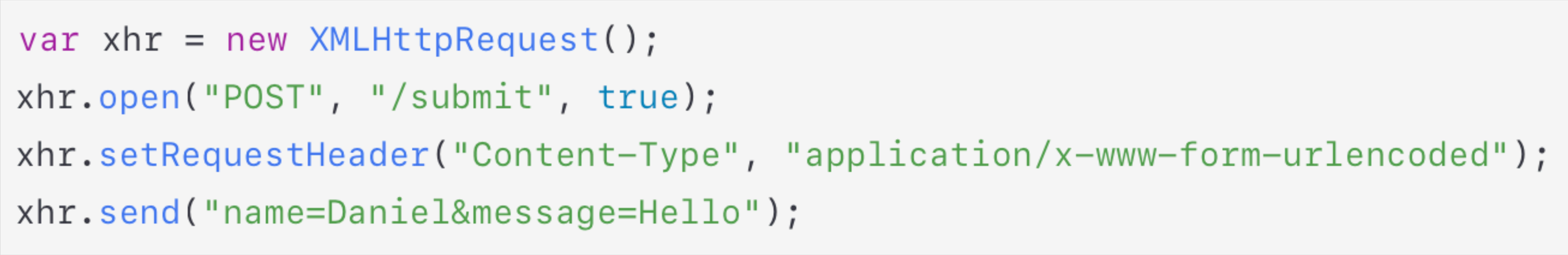


client is **blocked** while waiting for server response

AJAX: asynchronous JavaScript and XML



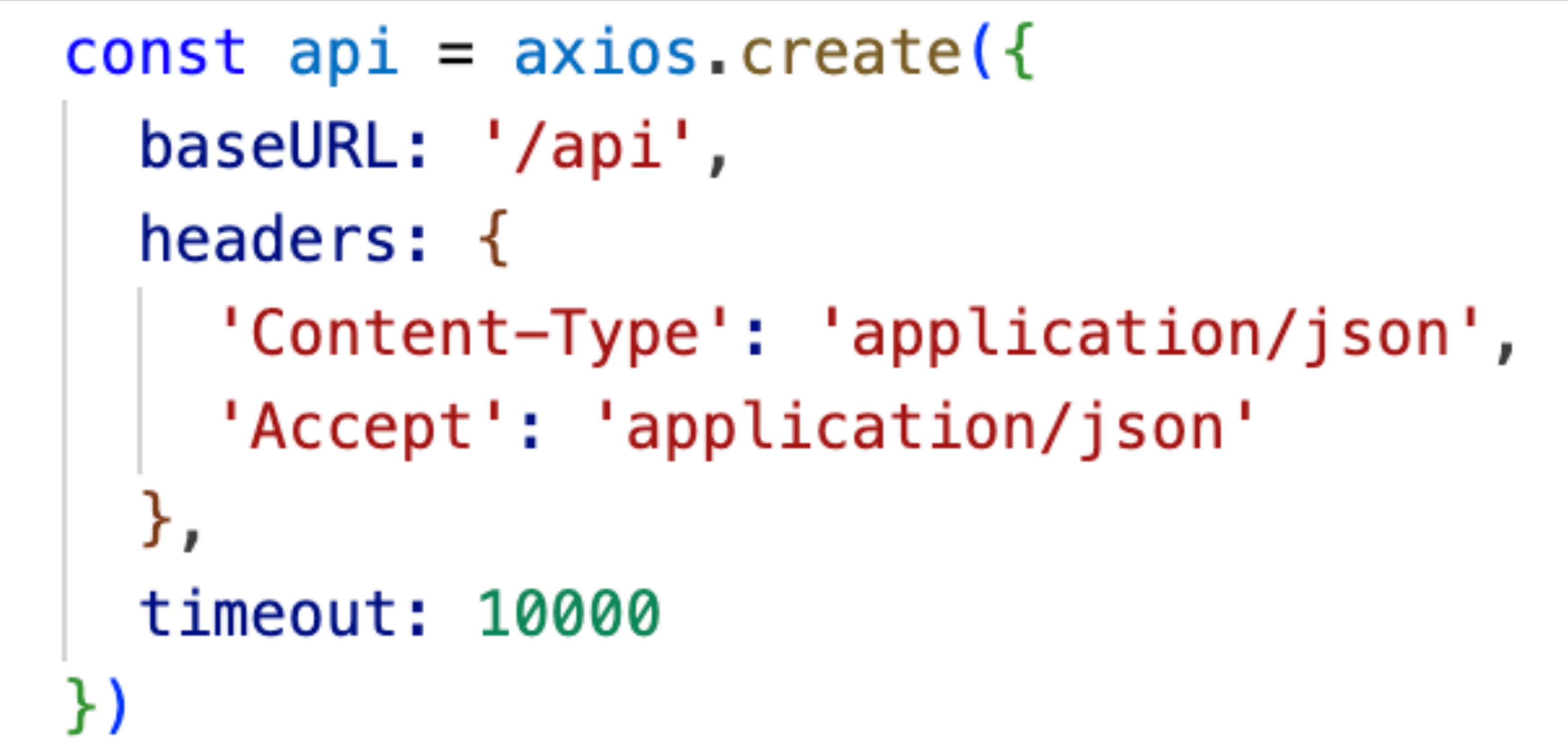
```
var xhr = new XMLHttpRequest();
xhr.open("GET", "example.txt", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log(xhr.responseText);
  }
};
xhr.send();
```



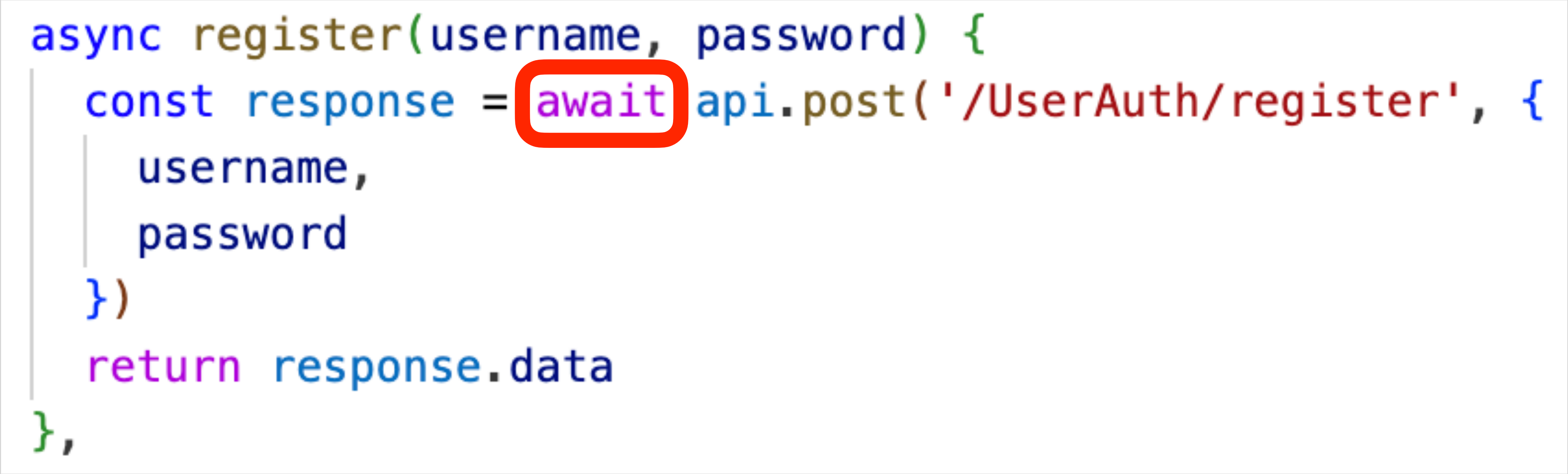
```
var xhr = new XMLHttpRequest();
xhr.open("POST", "/submit", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("name=Daniel&message=Hello");
```

XMLHttpRequest (1998)
calling server inside a script in the browser
introduced by Microsoft
later standardized for all browsers

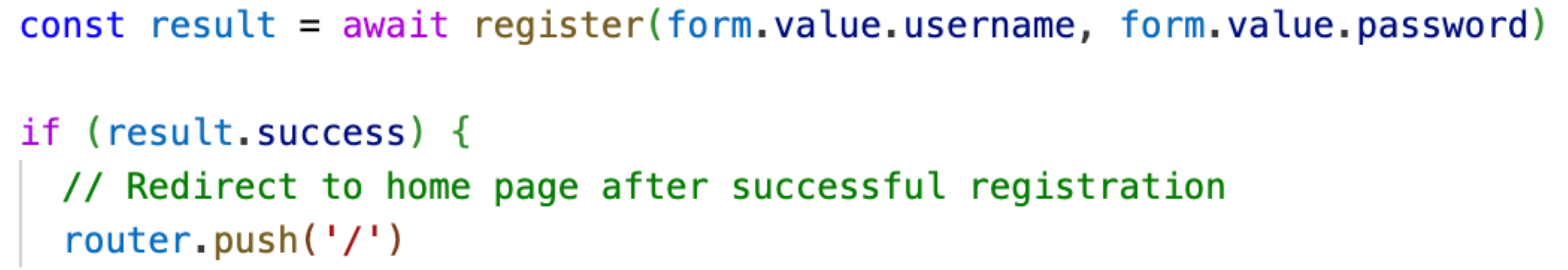
what server calls look like now



```
const api = axios.create({
  baseURL: '/api',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
  },
  timeout: 10000
})
```



```
async register(username, password) {
  const response = await api.post('/UserAuth/register', {
    username,
    password
  })
  return response.data
},
```



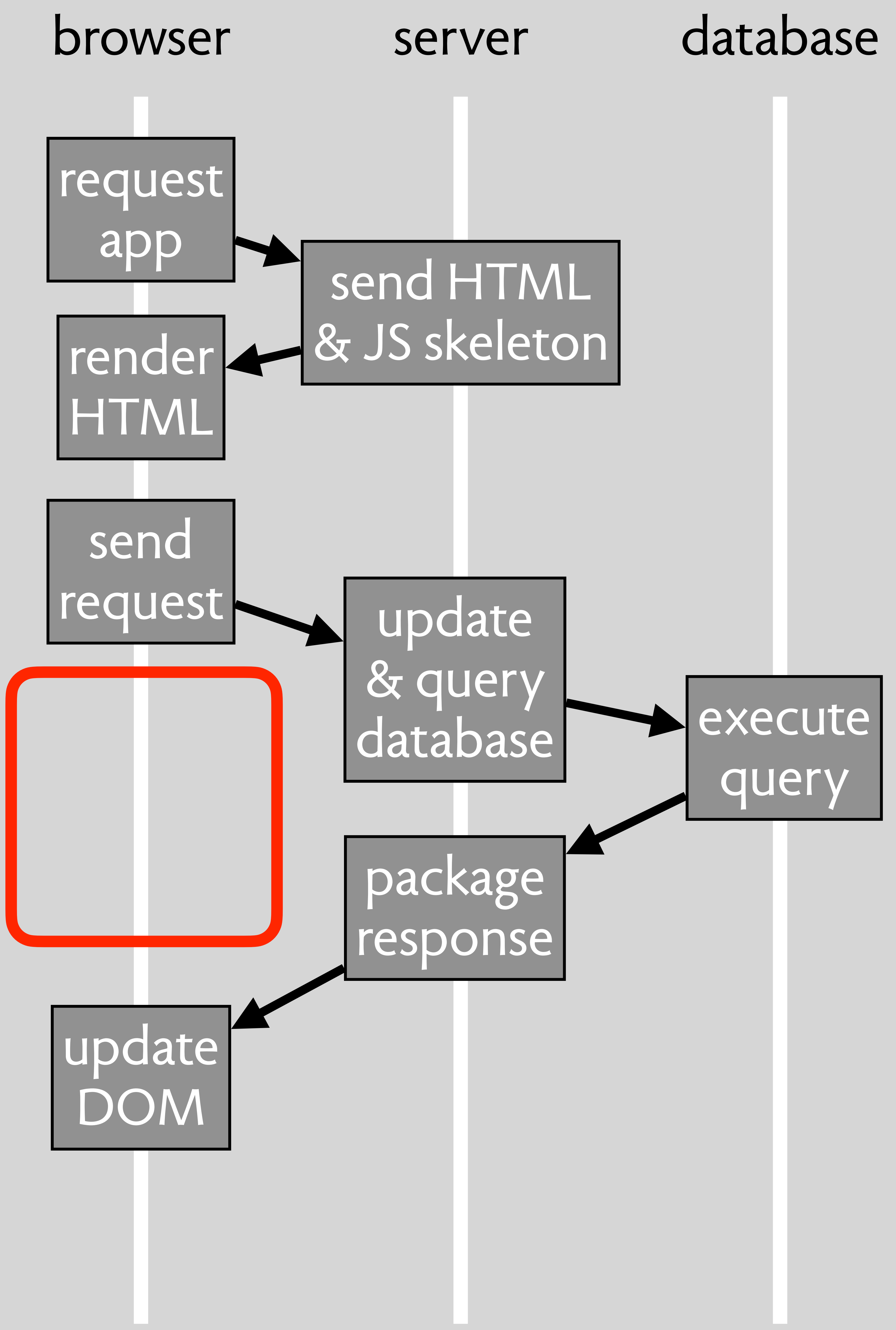
```
const result = await register(form.value.username, form.value.password)

if (result.success) {
  // Redirect to home page after successful registration
  router.push('/')
```

await
semantically, a blocking call
as if this “thread” waits for return
but other events still processed

now like a local call
but asynchronous

the flow for a “single page” app



client is **not blocked** while waiting for server response

what goes on
the client or server?

security considerations

code and data in the browser
are visible to and modifiable by the user
with developer tools

user can issue any HTTP requests
by modifying JS in the browser document
by commands in the browser JS console
by using curl or Postman

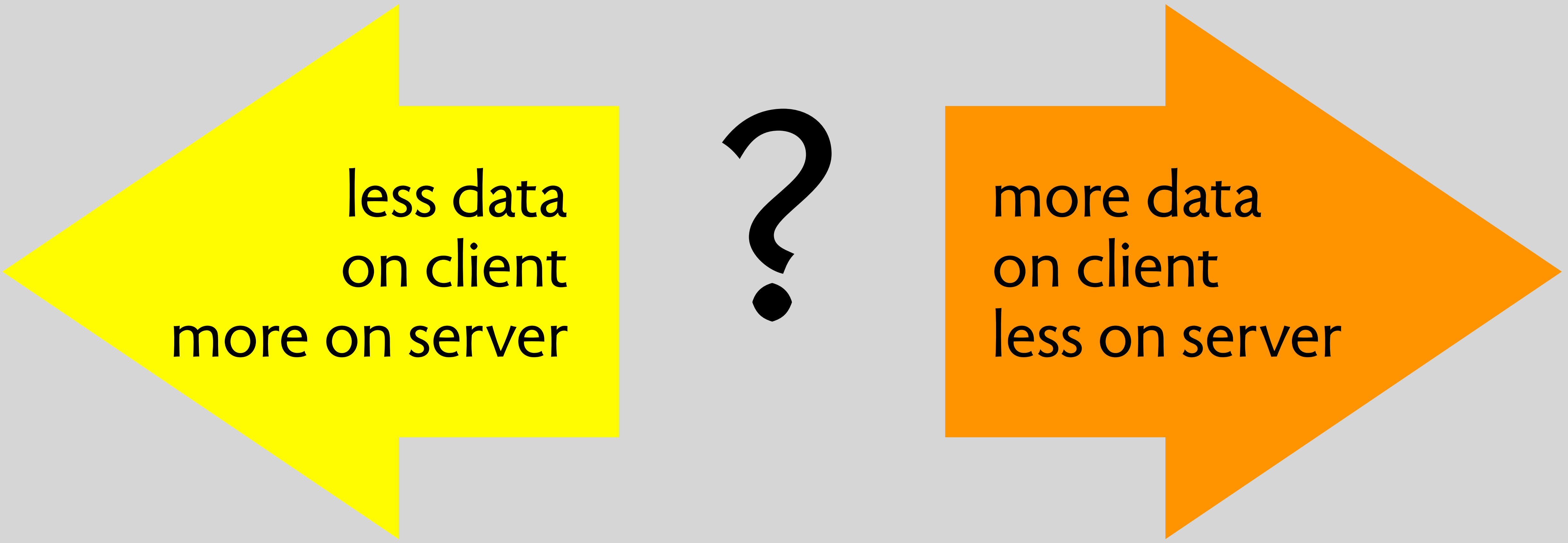
so which of these are good strategies?

to prevent access to another user's data
have client code pass user name with request ✗

to prevent access to sensitive pages
navigate first through login page ✗

to prevent access to another user's data
autoincrement session ids and store in cookie ✗

to prevent access to another user's data
generate random session id and store in cookie ✓

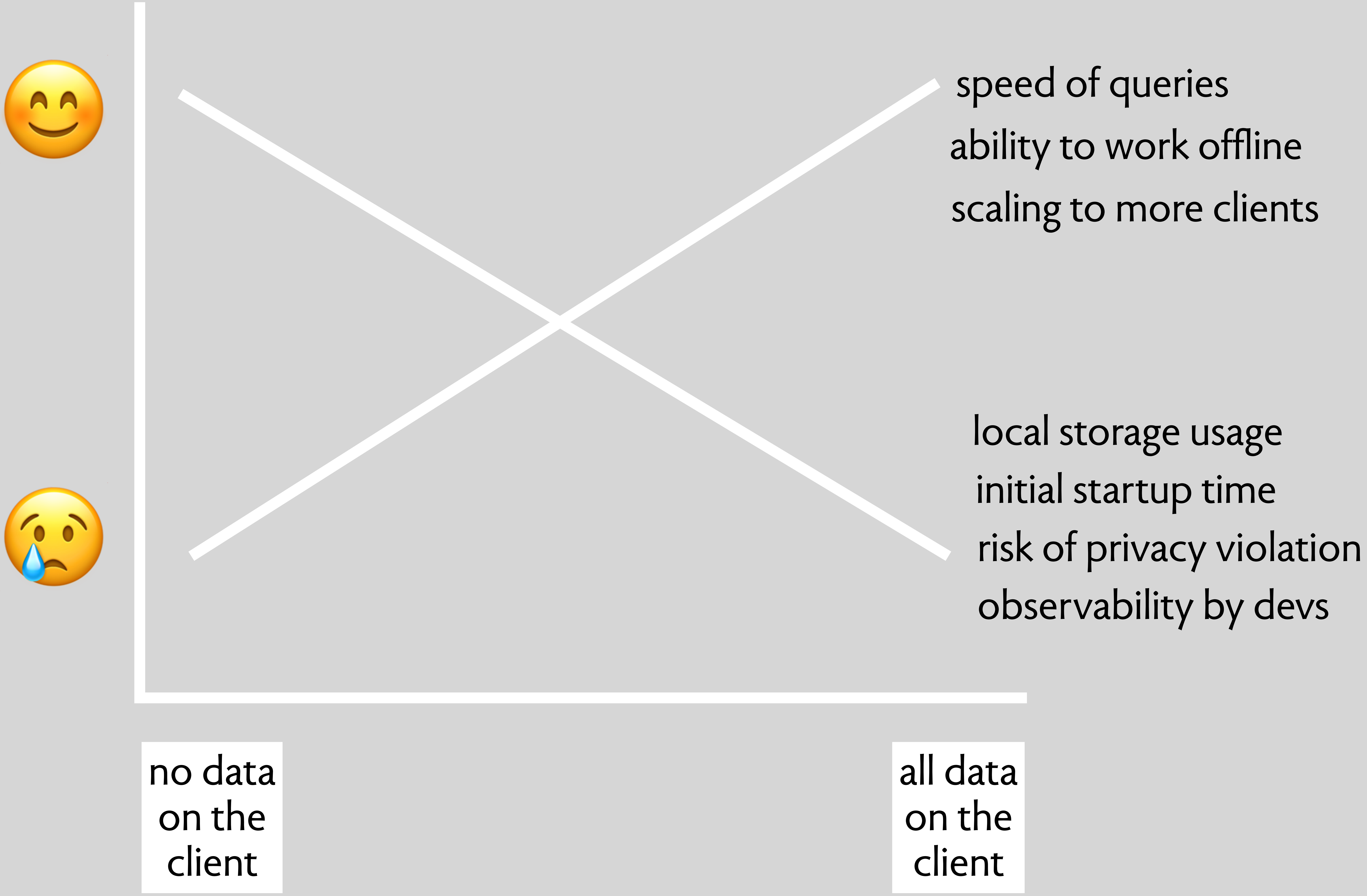


less data
on client
more on server

?

more data
on client
less on server

performance considerations



speed of queries

ability to work offline

scaling to more clients

local storage usage

initial startup time

risk of privacy violation

observability by devs

no data on the client

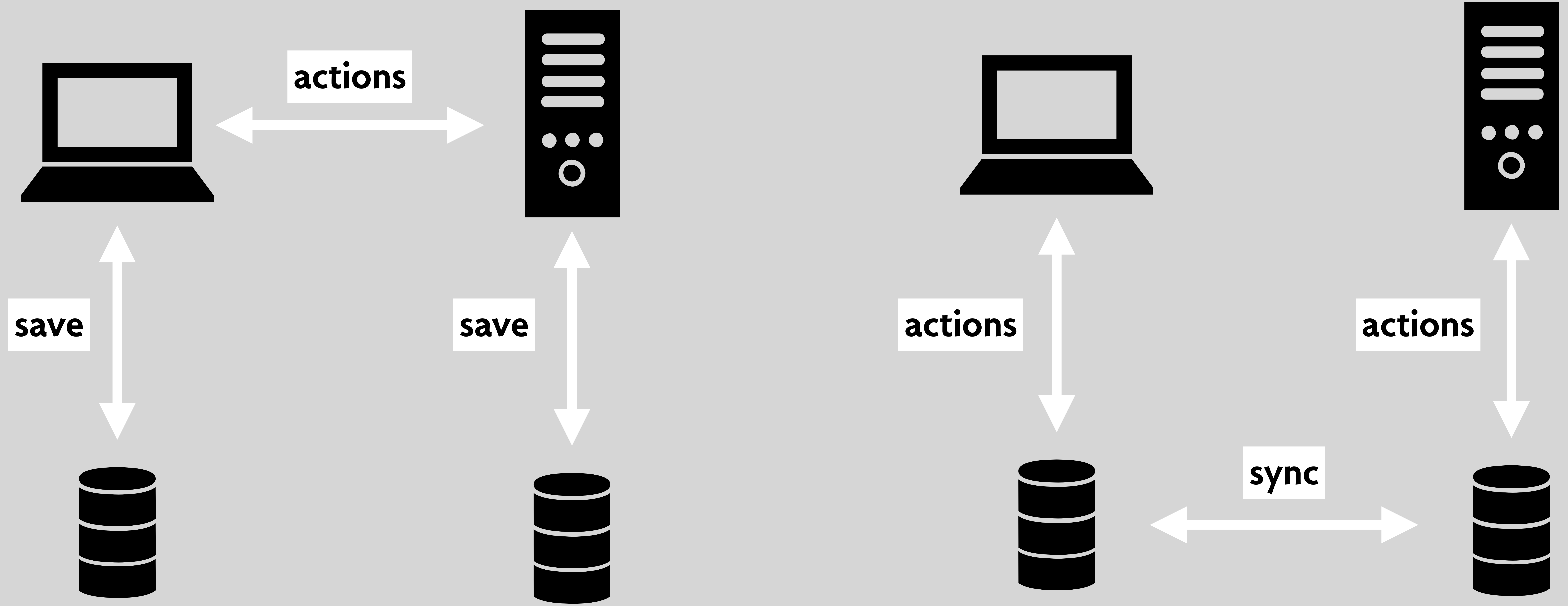
all data on the client

online apps are slow!

to Tokyo
144 ms
84 ms
108 ms
278 ms
144 ms
to California
234 ms
163 ms
240 ms
90 ms
301 ms
301 ms
144 ms
to São Paulo
243 ms
363 ms
334 ms

server-to-server round trip times between AWS data centers (Ink & Switch)

local first: a proposal for a new kind of app



full offline function
very fast read/write
user owns data
peer-to-peer too

stale data
collaboration hard
conflicts on sync

beyond iteration

a program × 3 ways

typescript

arrays functionals

a programming problem

```
interface User {
  name: string;
  active: boolean;
  purchases: number[];
}

const users: User[] = [
  { name: "Alice",   active: true,  purchases: [23, 19] },
  { name: "Bob",     active: false, purchases: [12] },
  { name: "Charlie", active: true,  purchases: [50, 10] },
  { name: "Dina",    active: true,  purchases: [] },
];
```

make an array of the active users with their purchase totals, like this:

```
[
   { name: "Alice", total: 42 },
   { name: "Charlie", total: 60 }
]
```

a conventional solution

```
interface Row { name: string; total: number };

const result: Row[] = [];

for (const user of users) {
  if (!user.active || user.purchases.length == 0) continue;
  let total = 0;
  for (const amount of user.purchases) {
    total += amount;
  }
  result.push({ name: user.name, total });
}
```

what's good or bad?
familiar constructs
but structure of the
function obscured

the classic list functionals

```
function filter<T>(a: T[], predicate: (e: T) => boolean): T[] {
    const result: T[] = [];
    for (let i = 0; i < a.length; i++)
        if (predicate(a[i]))
            result.push(a[i]);
    return result;
    }
```

```
const a: number[] = [1, 2, 3];
console.log (filter (a, e => e % 2 === 1));
// [1, 3]
```

```
function map<T, U>(a: T[], f: (e: T) => U): U[] {
    const result: U[] = [];
    for (let i = 0; i < a.length; i++)
        result.push(f(a[i]));
    return result;
    }
```

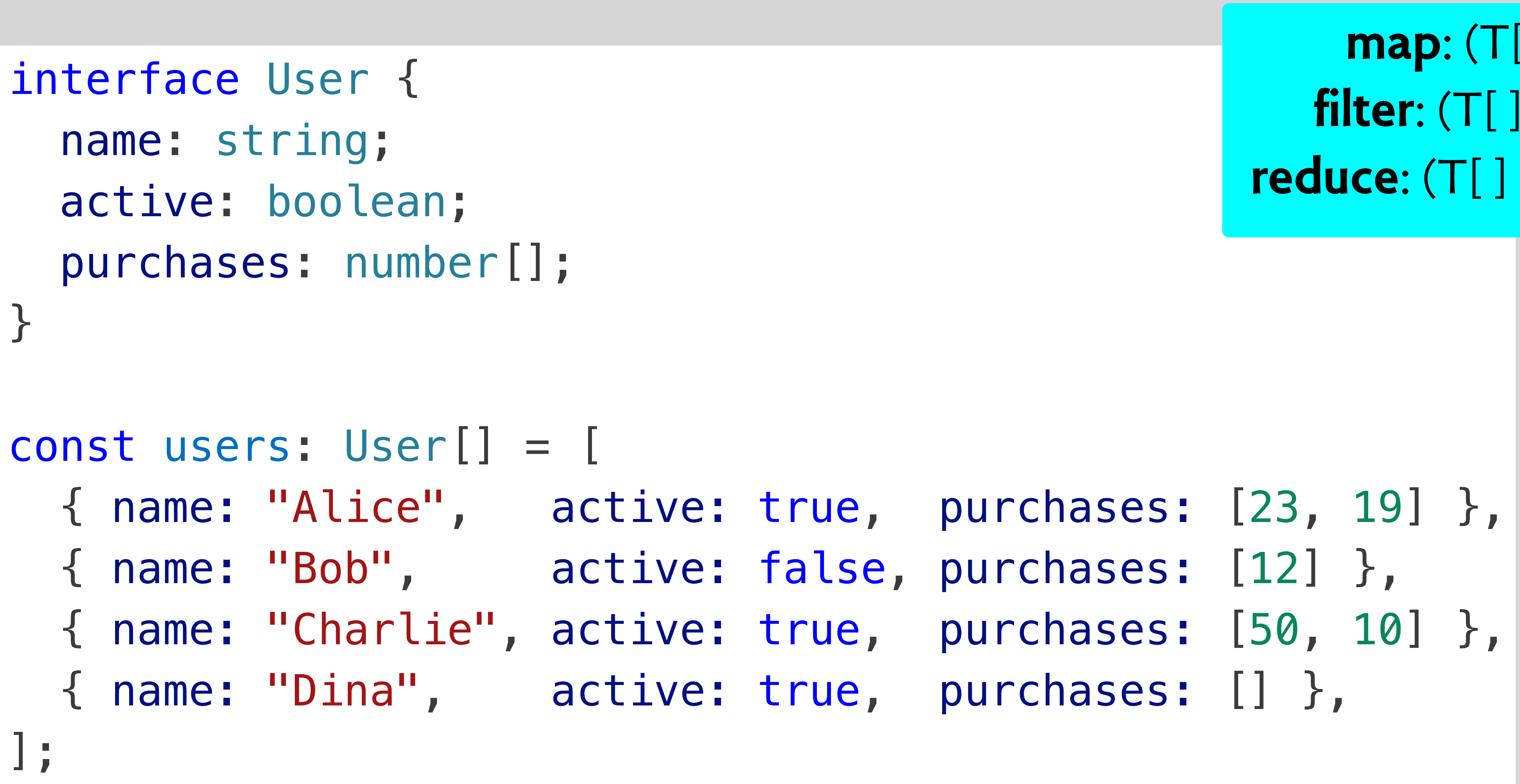
```
console.log (map (a, x => x * 2));
// [2, 4, 6]
```

```
function reduce<T>(a: T[], f: (acc: T, e: T) => T, init: T): T {
    let acc: T = init;
    for (let i = 0; i < a.length; i++)
        acc = f(acc, a[i]);
    return acc;
    }
```

```
console.log (reduce (a, (acc, e) => acc + e, 0));
// 6
```

can you see which functionals might be used for this?

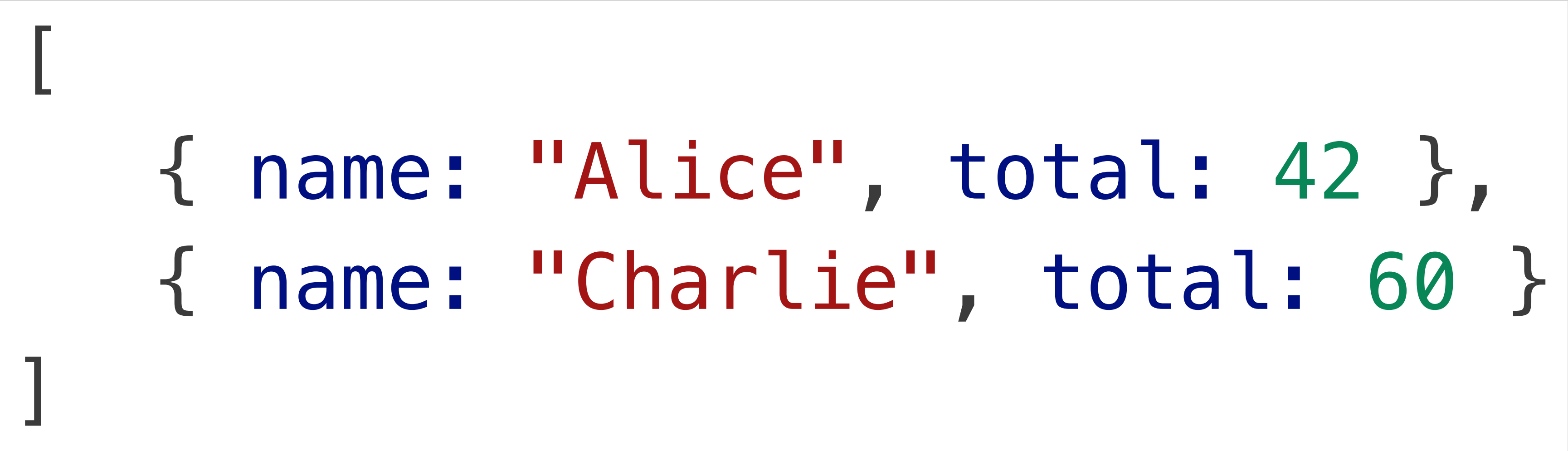
map: (T[ ] , T -> U) -> U [ ]
filter: (T[ ] , T -> bool) -> T [ ]
reduce: (T[ ] , (T, U) -> U, U) -> T [ ]



```
interface User {
  name: string;
  active: boolean;
  purchases: number[];
}

const users: User[] = [
  { name: "Alice",   active: true,  purchases: [23, 19] },
  { name: "Bob",     active: false, purchases: [12] },
  { name: "Charlie", active: true,  purchases: [50, 10] },
  { name: "Dina",    active: true,  purchases: [] },
];
```

make an array of the active users with their purchase totals, like this:



```
[
   { name: "Alice", total: 42 },
   { name: "Charlie", total: 60 }
]
```

rewriting our program with functionals

```
// get users with totals of their purchases, active users only
const activeUserPurchaseTotals = map(
  filter(users, u => u.active),
  (u => ({name: u.name,
          total: reduce(u.purchases, (acc, p) => acc + p, 0)
         }))
  );
```

```
console.log(activeUserPurchaseTotals);
/*
[
  { name: "Alice",   total: 42 },
  { name: "Charlie", total: 60 },
  { name: "Dina",    total: 0 },
]
*/
```

another example

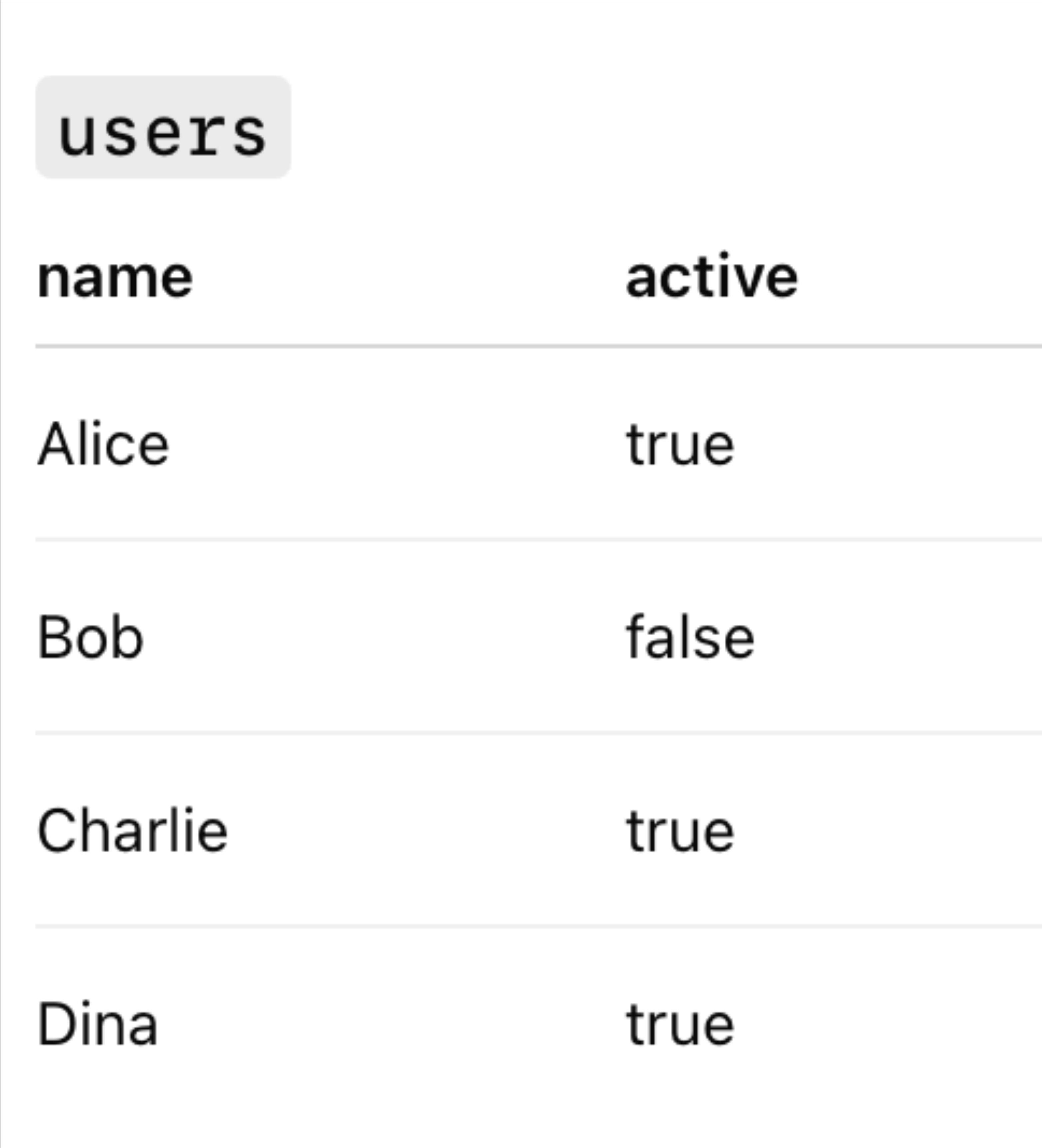
```
// get names of users with purchases over 30
const bigSpenders: string[] = map(
  filter(users, u => reduce(u, (acc, p) => acc + p, 0) > 30),
  u => u
);
```

```
console.log(bigSpenders); // ["Alice", "Charlie"]
```

SQL

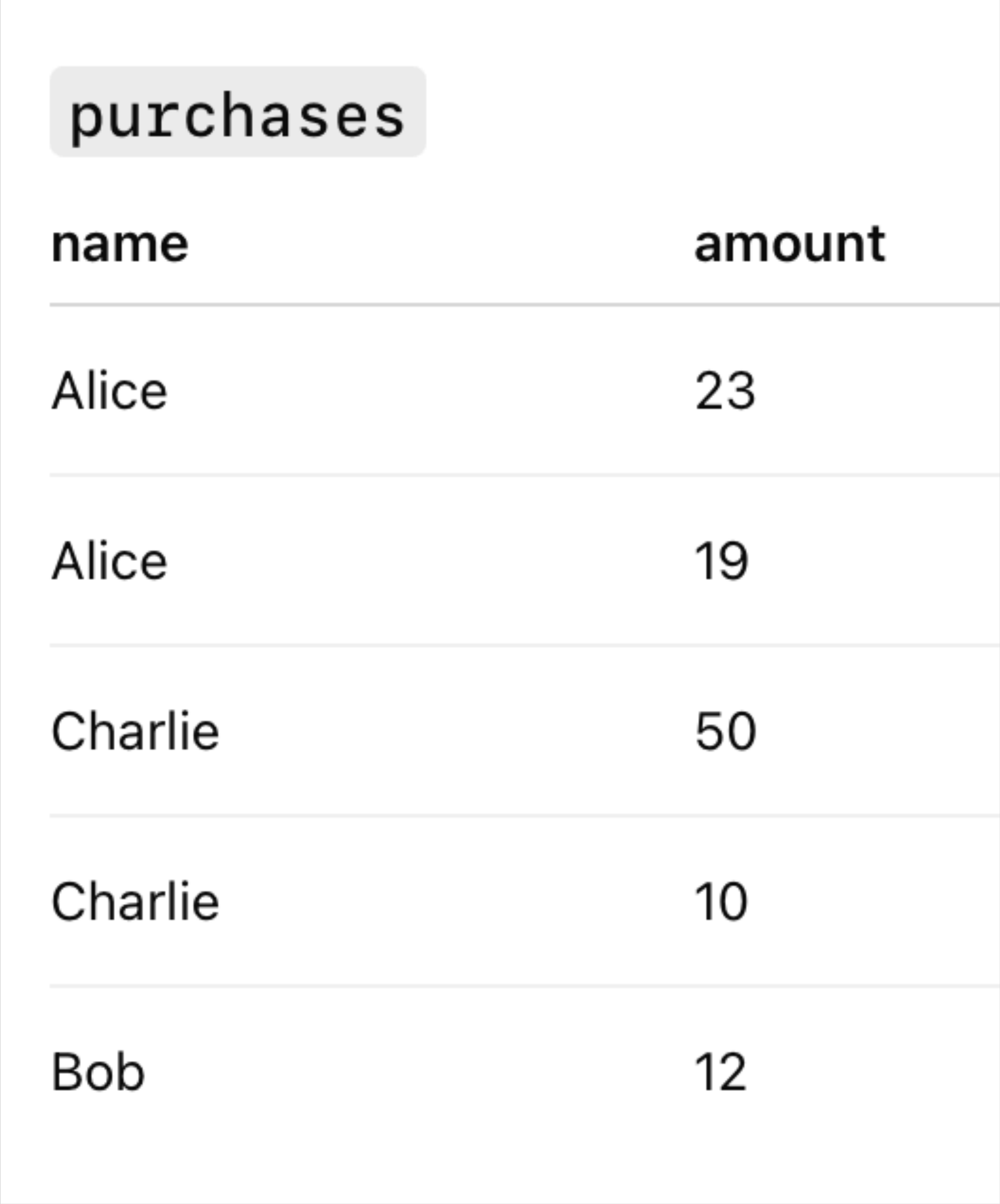
relational operators

represent data as tables of scalars



`users`

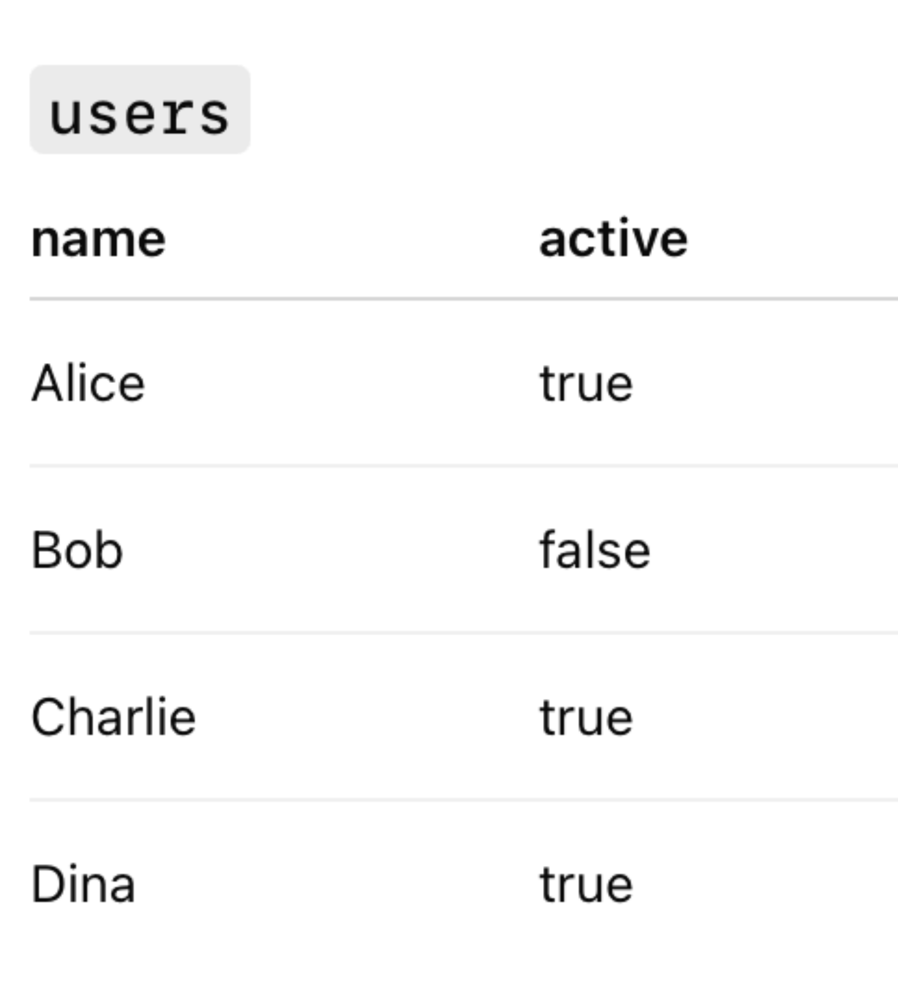
name	active
Alice	true
Bob	false
Charlie	true
Dina	true



`purchases`

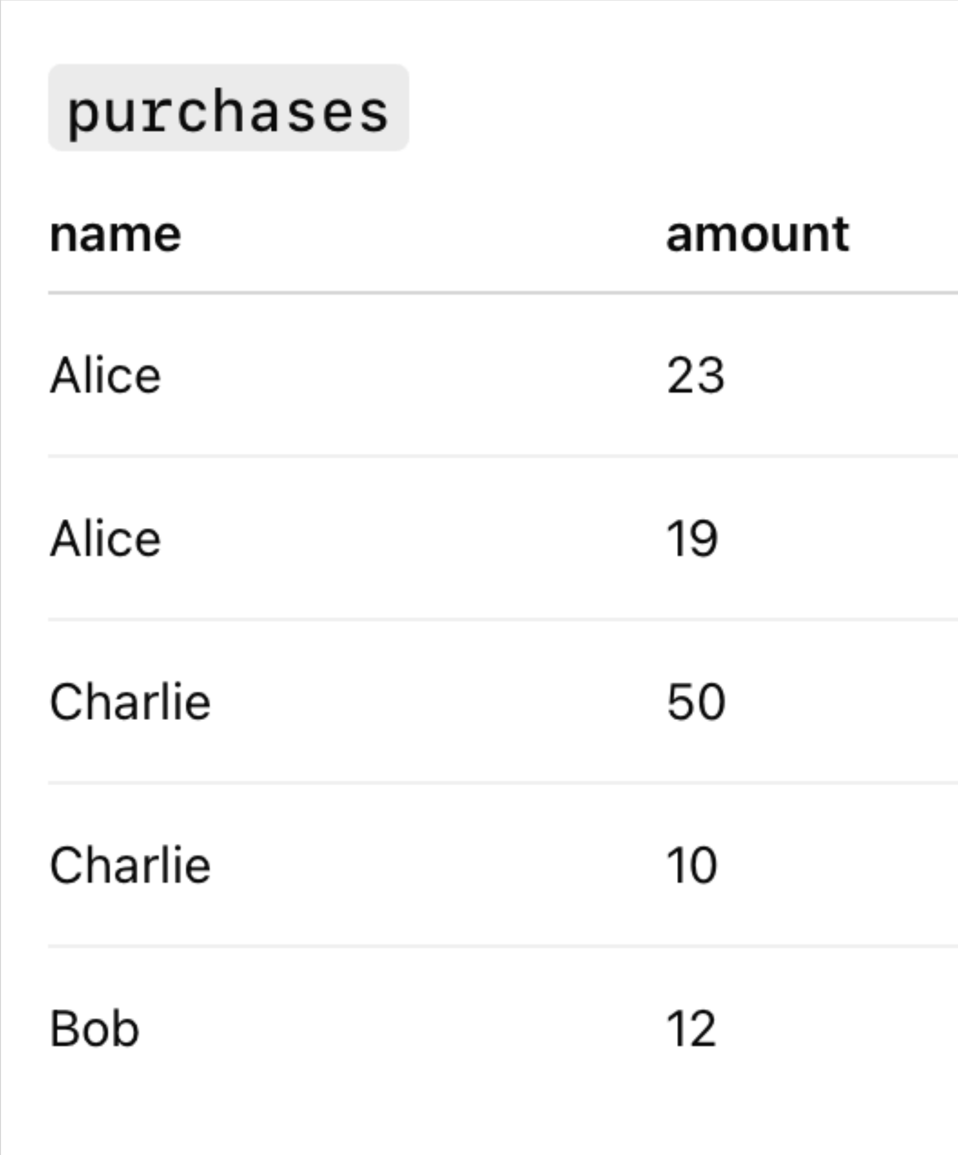
name	amount
Alice	23
Alice	19
Charlie	50
Charlie	10
Bob	12

joining tables



`users`

name	active
Alice	true
Bob	false
Charlie	true
Dina	true

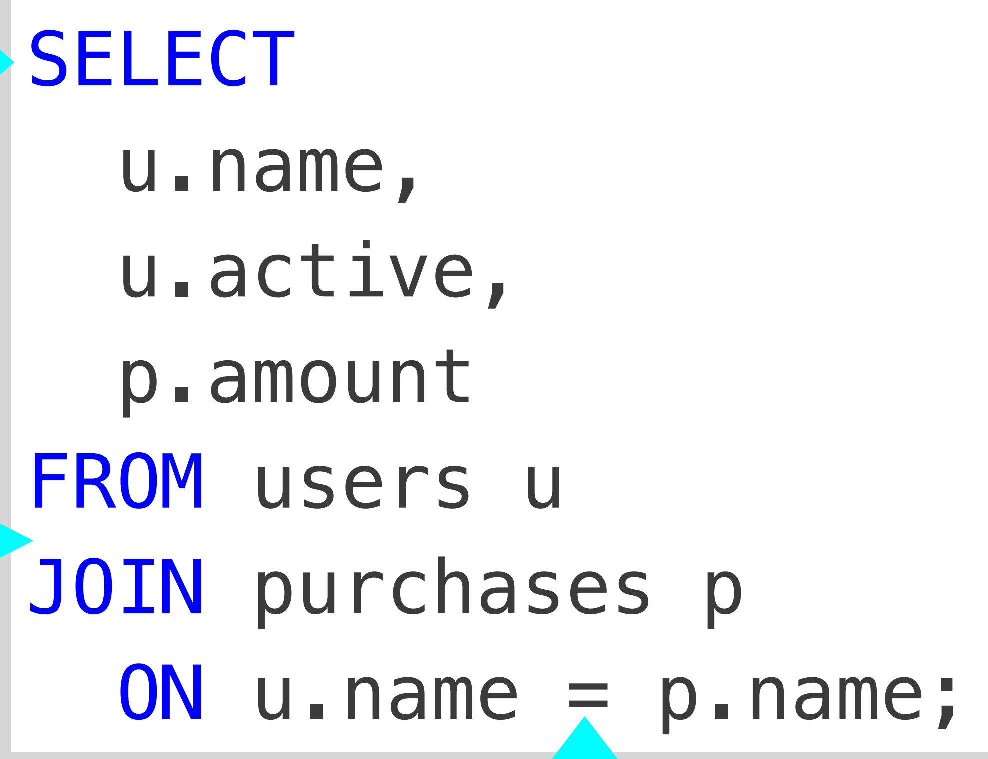


`purchases`

name	amount
Alice	23
Alice	19
Charlie	50
Charlie	10
Bob	12

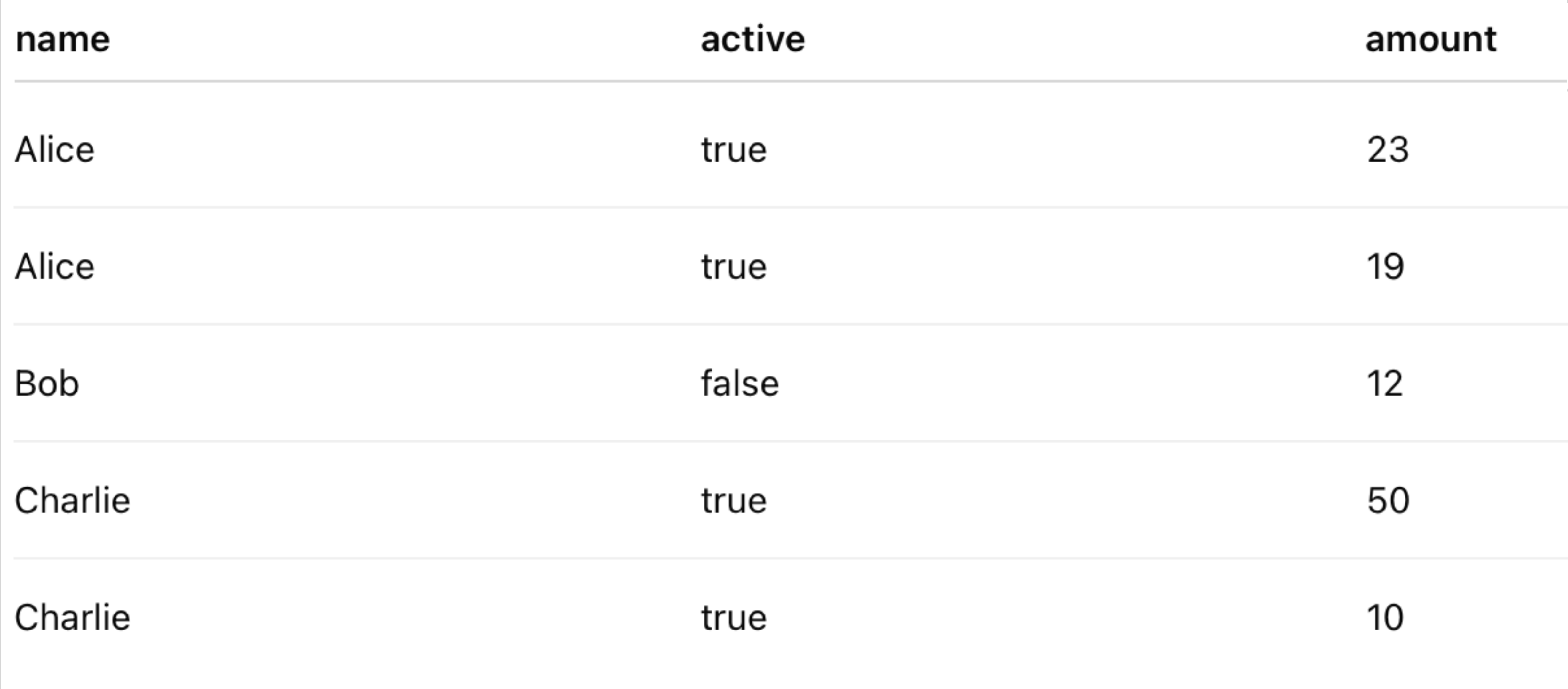
drop all columns except these

take all row combinations from the two tables



```
SELECT
  u.name,
  u.active,
  p.amount
FROM users u
JOIN purchases p
  ON u.name = p.name;
```

keep only new rows where names match



name	active	amount
Alice	true	23
Alice	true	19
Bob	false	12
Charlie	true	50
Charlie	true	10

restricting to active users

users

name	active
Alice	true
Bob	false
Charlie	true
Dina	true

purchases

name	amount
Alice	23
Alice	19
Charlie	50
Charlie	10
Bob	12

```
SELECT
  u.name,
  u.active,
  p.amount
FROM users u
JOIN purchases p
  ON u.name = p.name
WHERE u.active = TRUE;
```

keep only rows for active users

name	active	amount
Alice	true	23
Alice	true	19
Charlie	true	50
Charlie	true	10

summing purchases

`users`

name	active
Alice	true
Bob	false
Charlie	true
Dina	true

`purchases`

name	amount
Alice	23
Alice	19
Charlie	50
Charlie	10
Bob	12

```sql
SELECT
  u.name,
  SUM(p.amount) AS total
FROM users u
JOIN purchases p ON p.name = u.name
WHERE u.active = TRUE
GROUP BY u.name
ORDER BY u.name;
```

sum within each group

group rows by name

sort by name

name	total
Alice	42
Charlie	60

comparing SQL to list functionals

similar spirit

operations over lists
operations over rows

similar functions

list filter is like SQL where
list reduce is like SQL aggregates
list map can compute join

MongoDB collection queries

with normalized collections

```
// users collection
interface UserDoc {
  _id: string;
  name: string;
  active: boolean;
}

// purchases collection
interface PurchaseDoc {
  _id: string;
  name: string;
  amount: number;
}
```

```
db.users.aggregate([
  { $match: { active: true } },
  {
    $lookup: {
      from: "purchases",
      localField: "name",
      foreignField: "name",
      as: "purchases"
    }
  },
  { $unwind: "$purchases" },
  {
    $group: {
      _id: "$name",
      total: { $sum: "$purchases.amount" }
    }
  },
  { $project: { _id: 0, name: "$_id", total: 1 } },
  { $sort: { name: 1 } }
]);
```

filter

join

breaks up arrays

aggregate op

rename

sort

with a nested collection

```
interface UserEmbeddedDoc {
  _id: string;
  name: string;
  active: boolean;
  purchases: { amount: number }[];
}
```

this seems simpler

so why prefer the normalized version?

two reasons
separation of concerns
conflicts & locking

```
db.users.aggregate([
  { $match: { active: true } },
  { $unwind: "$purchases" },
  {
    $group: {
      _id: "$name",
      total: { $sum: "$purchases.amount" }
    }
  },
  { $project: { _id: 0, name: "$_id", total: 1 } },
  { $sort: { name: 1 } }
]);
```

using queries in a synchronization

```
export const GetActiveUserPurchaseTotals: Sync = (
  { request, user, username, total, results } ) => ({
  when: actions(
    [Requesting.request, { path: "/purchase-totals" }, { request }]
    ),
  where: async (frames) => {
    frames = await frames.query(User._getActiveUsers, {}, { user });
    frames = await frames.query(User._getUsername, { user }, { username });
    return await frames.query(Purchasing._getTotalForUser, { user }, { total });
    },
  then: actions(
    [Emailing.email, { user, username, total }]
    )
  });
```

declare sync vars

```
[{request: ..}]

[{request: .., user: ..},
 {request: .., user: ..}, ..]

[{request: ..,
  user: ..,
  username: ..}, ..
]

[{request: ..,
  user: ..,
  username: ..,
  total: .. }, ..
]
```

```
concept User
state
  a set of Users with
    a username String
    an active Boolean = true
queries
  _getActiveUsers (): (user: User)
   effects returns set of active Users

  _getUsername (user: User): (username: String)
   effects returns username associated with user
```

```
concept Purchasing
state
  a set of Purchases with
    user: User
    reason: String
    amount: Number
queries
   _getTotalForUser (user: User): (total: Number)
   effects returns the sum of the amount
      for all user’s purchases
```

how to collect results for single response

```
export const GetActiveUserPurchaseTotals: Sync = (
  { request, user, username, total, results } ) => ({
  when: actions(
    [Requesting.request, { path: "/purchase-totals" }, { request }]
    ),
  where: async (frames) => {
    frames = await frames.query(User._getActiveUsers, {}, { user });
    frames = await frames.query(User._getUsername, { user }, { username });
    frames = await frames.query(Purchasing._getTotalForUser, { user }, { total });
    return frames.collectAs([user, username, total], results);
    },
  then: actions(
    [Requesting.respond, { request, results }]
    )
  });
```

```
[{request: ..}]

[{request: .., user: ..},
 {request: .., user: ..}, ..]

[{request: ..,
  user: ..,
  username: ..}, ..
]

[{request: ..,
  user: ..,
  username: ..,
  total: .. }, ..
]

[{results:

   [{user: ..,
     username: ..,
     total: .. }, ..
   ]

}]
```

```
concept User
state
  a set of Users with
    a username String
    an active Boolean = true
queries
  _getActiveUsers (): (user: User)
   effects returns set of active Users

  _getUsername (user: User): (username: String)
   effects returns username associated with user
```

```
concept Purchasing
state
  a set of Purchases with
    user: User
    reason: String
    amount: Number
queries
   _getTotalForUser (user: User): (total: Number)
   effects returns the sum of the amount
      for all user's purchases
```

what Eagon is going to show you

how your back-end code is organized
you write ONLY concept and sync files
(and configure inclusions & exclusions)

what the framework provides (you can ignore this)
sync engine: handles event/data flow, tracing, etc
Requesting concept: encapsulates HTTP

where to find background files
for you and the LLM to read

how to work with syncs
including tracing to console