GraphQL from schema to production

You build a GraphQL API with Apollo, from the first schema to a public graph that limits cost and can evolve safely.

Mike Bild

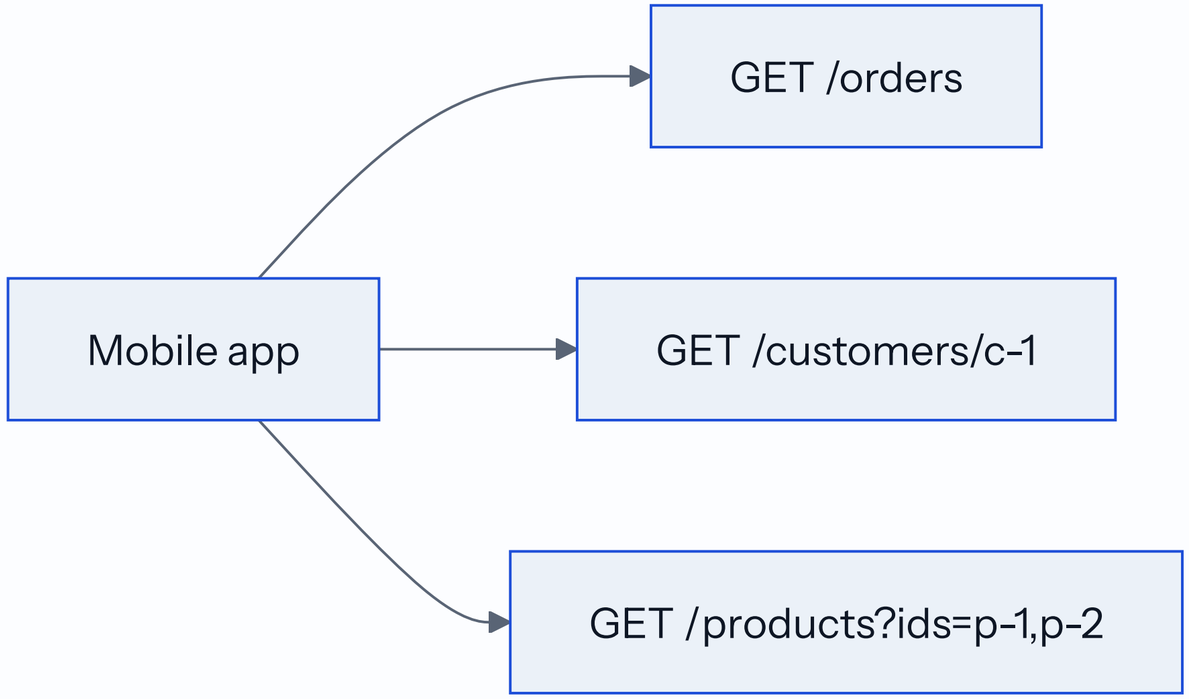
Exercises: github.com/MikeBild/introduction-graphql

Basics and first server

Why GraphQL? · Every exclamation point in the schema is a promise · First server with Apollo Server and Express · Operations instead of assembled query strings

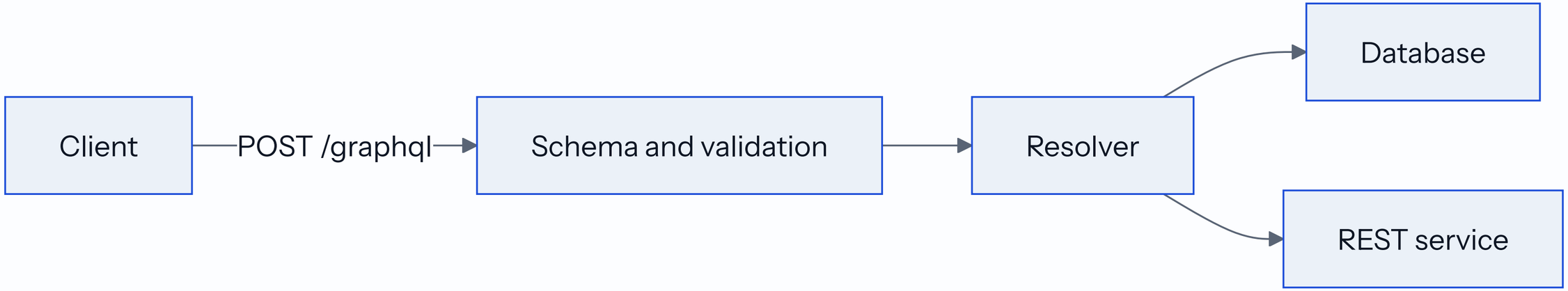
Fixed endpoints, many clients

- Many clients: web, app, and partners with their own views
- Overfetching: too many fields per response
- Underfetching: too many requests per view



One endpoint, one graph

- The server defines in the schema what may be requested
- The client selects exactly the fields it needs
- One Query replaces the request waterfall



The order overview as a Query

```
query OrderOverview {
  orders(status: PLACED) {
    id
    customer { name }
    items {
      quantity
      product { name priceCents }
    }
  }
}
```

When not?

- Simple CRUD with exactly one client
- Files, downloads, and CDN caching per URL
- No clear ownership for the schema

Your decision

- Many clients, connected data: GraphQL
- Resources, files, simple integrations: REST
- Both side by side is normal

A schema as a contract

- The schema describes what clients may ask for
- The server checks every operation before execution
- Tools read the schema through introspection

Object types and scalars

```
type Product {
  id: ID!
  name: String!
  priceCents: Int!
  stock: Int!
}

type Order {
  id: ID!
  status: OrderStatus!
  items: [OrderItem!]!
}

enum OrderStatus { DRAFT PLACED PAID SHIPPED CANCELLED }
```

Set nullability deliberately

Order
object or null

[Order!]
list may be null

[Order]!
items may be null

[Order!]!
never null

Query and Mutation

```
type Query {
  products: [Product!]!
  orders(status: OrderStatus): [Order!]!
  order(id: ID!): Order
}

type Mutation {
  cancelOrder(input: CancelOrderInput!): CancelOrderPayload!
}

input CancelOrderInput {
  orderId: ID!
}

type CancelOrderPayload {
  order: Order!
}
```

Where schema types are chosen badly

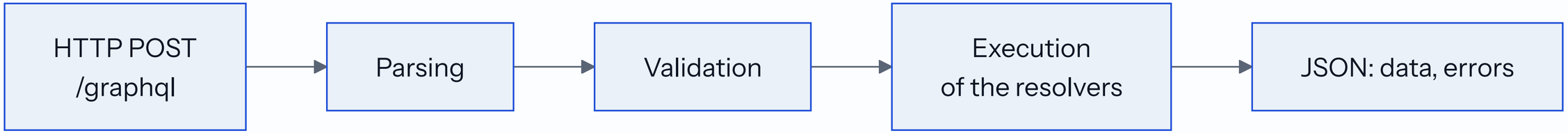
- Everything Non-Null: one error deletes half the response
- Everything nullable: clients check every field
- Strings instead of enums: typos become data

Exercise: the shop schema

- Product as an object type with price in cents
- Customer with Non-Null fields and an orders list
- Entry points under Query

What a server does

- Parse the query and validate it against the schema
- Run one resolver per field
- Return JSON with data and errors



Schema and first resolvers

```
const typeDefs = `#graphql
  enum OrderStatus { DRAFT PLACED PAID SHIPPED CANCELLED }
  type Order { id: ID!, status: OrderStatus! }
  type Query { orders: [Order!]! }
`

const orders = [{ id: 'o-1', status: 'PLACED' }]

const resolvers = {
  Query: {
    orders: () => orders,
  },
}
```

Apollo Server with Express

```
import express from 'express'
import { ApolloServer } from '@apollo/server'
import { expressMiddleware } from '@as-integrations/express5'

const server = new ApolloServer({ typeDefs, resolvers })
await server.start()

const app = express()
app.use('/graphql', express.json(), expressMiddleware(server))
app.listen(4000, () => console.info('GraphQL auf http://localhost:4000/graphql'))
```

Queries and responses

- Sandbox: autocomplete from the schema
- curl: POST with query and variables as JSON
- Field errors: status 200 with data and errors

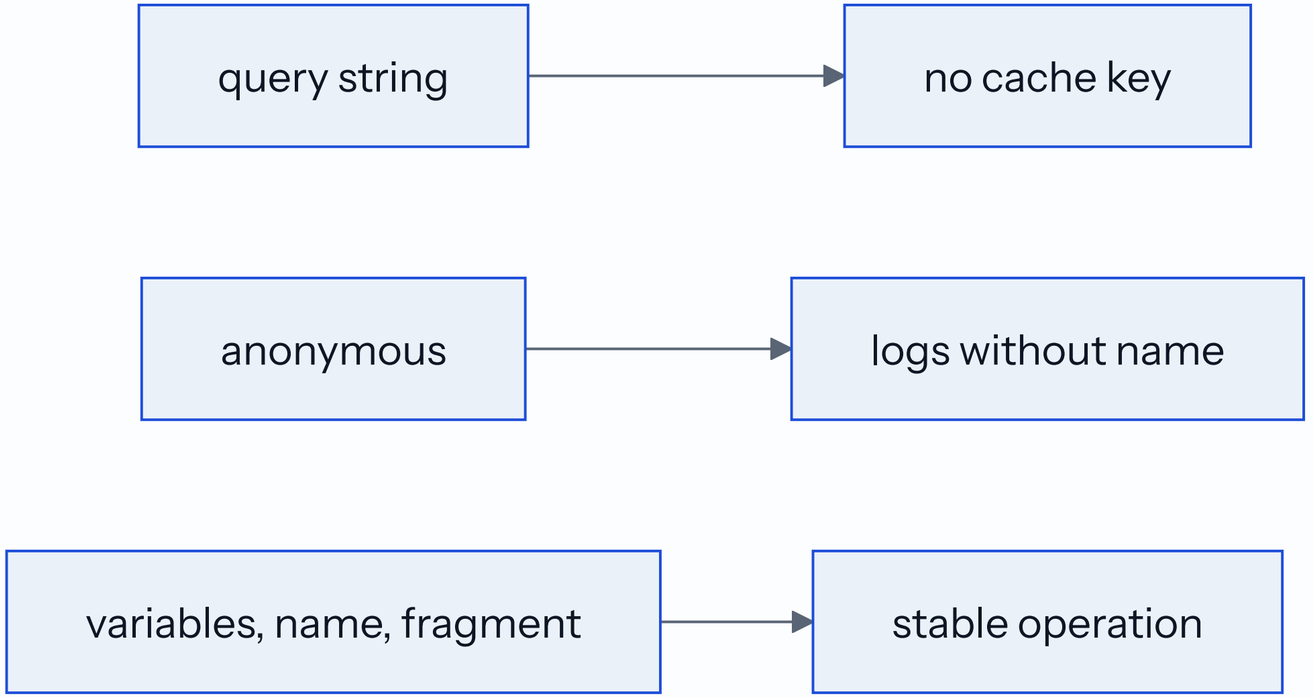
What breaks in the first server

- Schema and resolver drift apart
- any types in resolvers hide errors
- Internal error messages reach clients

Exercise: the shop responds

- Port the sample server to Apollo Server 5 and Express 5
- Add the resolver for Query.products
- Check the query with Sandbox and curl

Hard-coded queries



Variables

```
query OrderDetail($id: ID!) {
  order(id: $id) {
    id
    status
  }
}

# variables: { "id": "o-1" }
```

Fragments

```
fragment OrderRow on Order {
  id
  status
  totalCents
  customer { name }
}

query Orders {
  orders { ...OrderRow }
}
```

Aliases and directives

```
query OrderOverview($withItems: Boolean!) {
  placed: orders(status: PLACED) { ...OrderRow }
  paid: orders(status: PAID) {
    ...OrderRow
    items @include(if: $withItems) { quantity }
  }
}
```

Where operations fail validation

- Fragments must not form cycles
- Nullable variables do not match non-null arguments
- Multiple operations in a document need operationName

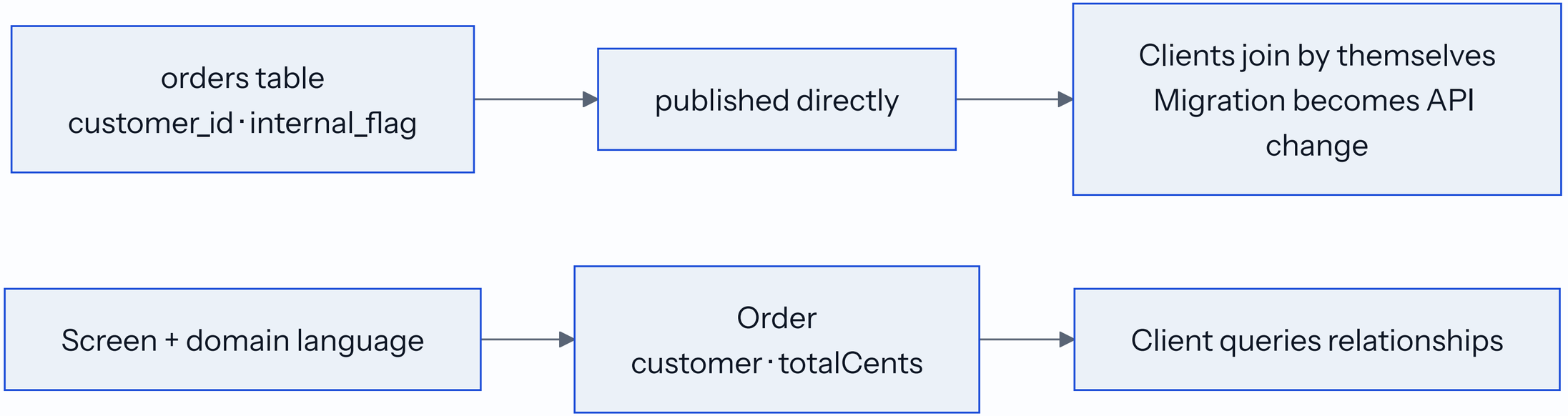
Exercise: the order overview

- Two aliases for PLACED and PAID
- One OrderRow fragment for both
- Toggle items with a directive

Schema, resolver, mutations, errors

Design schema first · Resolvers without N+1 · A mutation is a domain action · Model error handling in the schema

Tables are not an API model



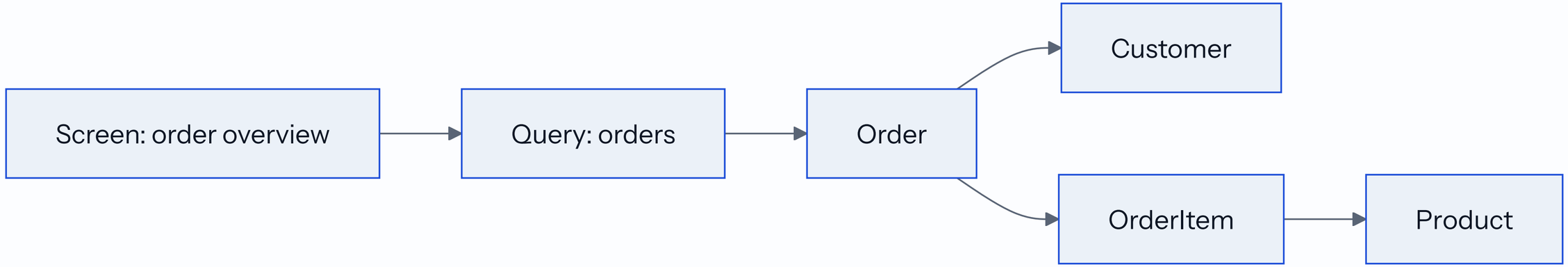
From domain term to type

```
"""Eine Bestellung, wie Kunden und Support sie sehen"""
type Order {
  id: ID!
  status: OrderStatus!
  customer: Customer!
  items: [OrderItem!]!
  totalCents: Int!
}

enum OrderStatus { DRAFT PLACED PAID SHIPPED CANCELLED }

type Query { order(id: ID!): Order }
```

Work backward from the screen



Rules that go far

- Justify nullability per field
- Declare fixed value sets as enums
- Shape mutations as domain actions, not updates

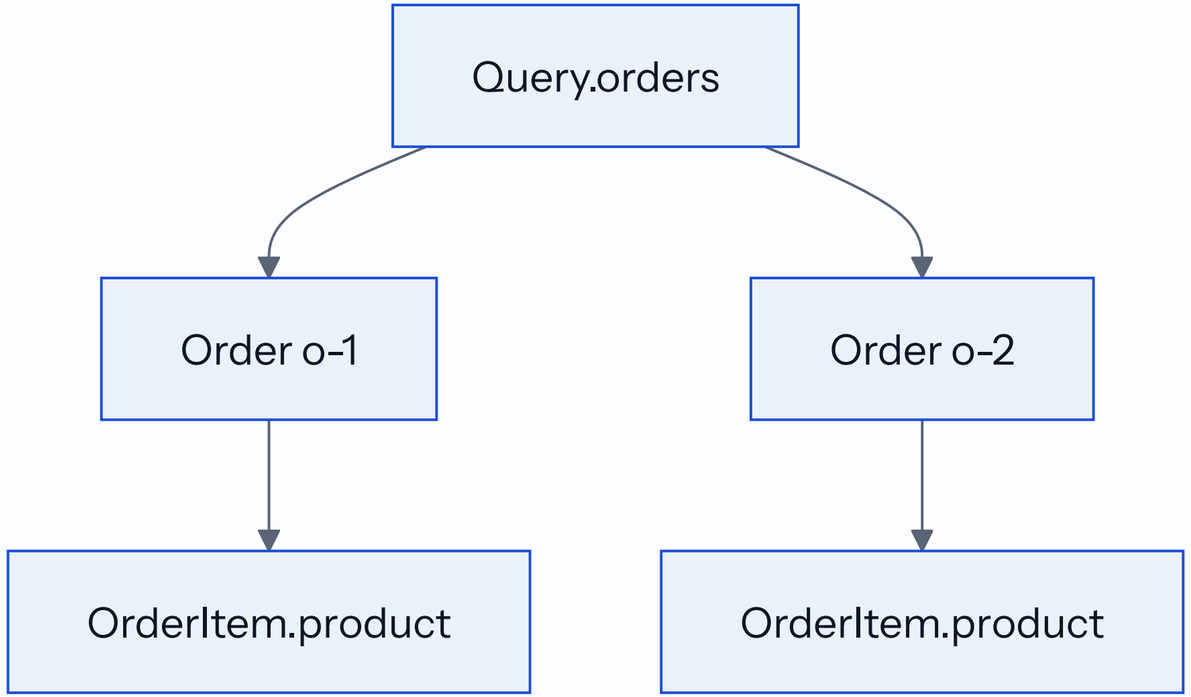
What makes a schema blurry

- Generic fields like data or payload as JSON
- God types with a hundred fields from all Bounded Contexts
- Fields in advance that nobody uses

Exercise: from screen to schema

- Sketch the screen and mark data
- Name types and relationships
- Add a Query and one mutation

One resolver per field



The N+1 problem

- One database access for the list
- N database accesses for the products
- N more accesses for every deeper level

```
const resolvers = {
  Query: { orders: () => db.orders.findMany() },
  OrderItem: {
    product: (item: OrderItemRow) => db.products.findById(item.productId),
  },
}
```

DataLoader batches

```ts
import DataLoader from 'dataloader'

export const createLoaders = () => ({
  product: new DataLoader(async (ids: readonly string[]) => {
    const rows = await db.products.findByIds(ids)
    const byId = new Map(rows.map((row) => [row.id, row]))
    return ids.map((id) => byId.get(id) ?? null)
  }),
})
```

One request, one loader

```
type Context = { loaders: ReturnType<typeof createLoaders> }

const server = new ApolloServer<Context>({ typeDefs, resolvers })
await server.start()

app.use('/graphql', express.json(), expressMiddleware(server, {
  context: async () => ({ loaders: createLoaders() }),
}))

// im Resolver
product: (item: OrderItemRow, _args: unknown, context: Context) =>
  context.loaders.product.load(item.productId)
```

Where DataLoader returns wrong data

- Global loader: data leaks between users
- Batch function: results in the wrong order
- After mutations: loader cache is stale

Exercise: count accesses

- Measure accesses for the order overview
- Add a loader for products in the context
- Compare the result: two accesses

CRUD via GraphQL

- An updateOrder has twenty optional fields
- No one knows which combination is allowed
- Domain rules move into the client

Domain actions instead of updates

- Actions are named placeOrder, cancelOrder, payOrder
- Each action checks its own rules
- The payload contains the changed object with its id

Input in, payload out

```
input OrderItemInput {
  productId: ID!
  quantity: Int!
}

input PlaceOrderInput {
  items: [OrderItemInput!]!
}

type PlaceOrderPayload {
  order: Order!
}

type Mutation {
  placeOrder(input: PlaceOrderInput!): PlaceOrderPayload!
}
```

The resolver stays thin

```
const resolvers = {
  Mutation: {
    placeOrder: async (_p: unknown, { input }: { input: PlaceOrderInput }, ctx: Context) => {
      const order = await orders.place(ctx.customerId, input)
      return { order }
    },
  },
}

// Business logic, independent of GraphQL
const place = async (customerId: string, input: PlaceOrderInput) => {
  if (input.items.length === 0) throw new Error('Mindestens eine Position')
  // Check inventory, apply prices, save
}
```

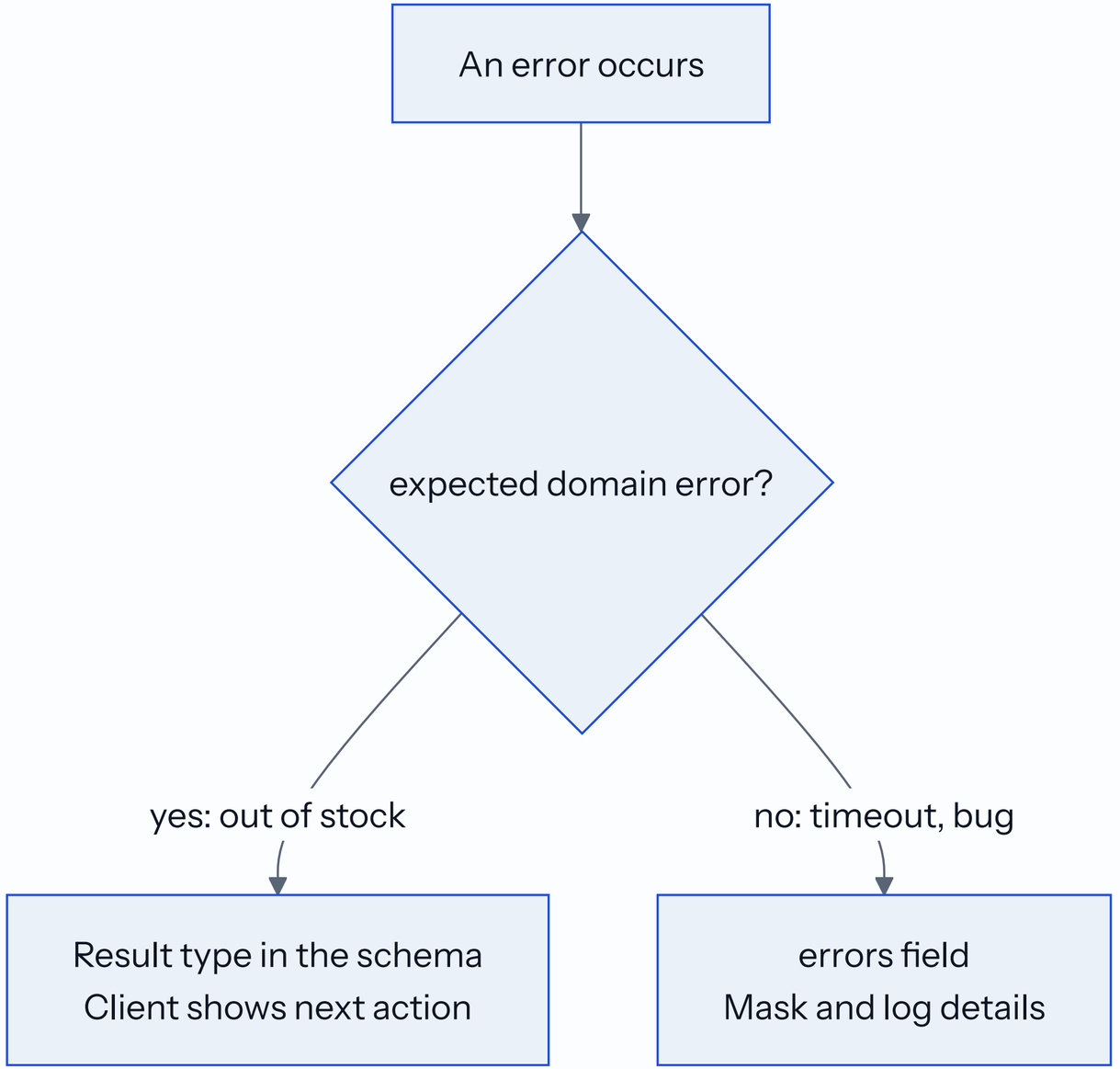
What makes mutations unreliable

- Multiple mutation fields run one after another, without a transaction
- Repeated requests need idempotency
- One input type for creating and changing gets fuzzy

Exercise: cancel an order

- Create CancelOrderInput and CancelOrderPayload
- Check the status rule in the domain logic
- Return the changed order in the payload

Two kinds of errors



The errors field

```
{
  "data": { "order": { "id": "o-1", "customer": null } },
  "errors": [{
    "message": "Unerwarteter Fehler",
    "path": ["order", "customer"],
    "extensions": { "code": "INTERNAL_SERVER_ERROR" }
  }]
}
```

Domain errors as types

```
union PlaceOrderResult = PlaceOrderSuccess | OutOfStock | EmptyOrder

type PlaceOrderSuccess { order: Order! }
type OutOfStock { productId: ID!, message: String! }
type EmptyOrder { message: String! }

type Mutation {
  placeOrder(input: PlaceOrderInput!): PlaceOrderResult!
}
```

Mask technical errors

```
import { unwrapResolverError } from '@apollo/server/errors'
import { GraphQLError, type GraphQLFormattedError } from 'graphql'

export const formatError = (formatted: GraphQLFormattedError, error: unknown) => {
  if (unwrapResolverError(error) instanceof GraphQLError) return formatted
  log.error(`[graphql] ${formatted.message}`)
  return { ...formatted, message: 'Internal server error' }
}

const server = new ApolloServer<Context>({ typeDefs, resolvers, formatError })
```

What makes error handling unusable

- Parsing error messages instead of checking error codes
- Exposing stack traces and SQL in the response
- Looking only at the HTTP status

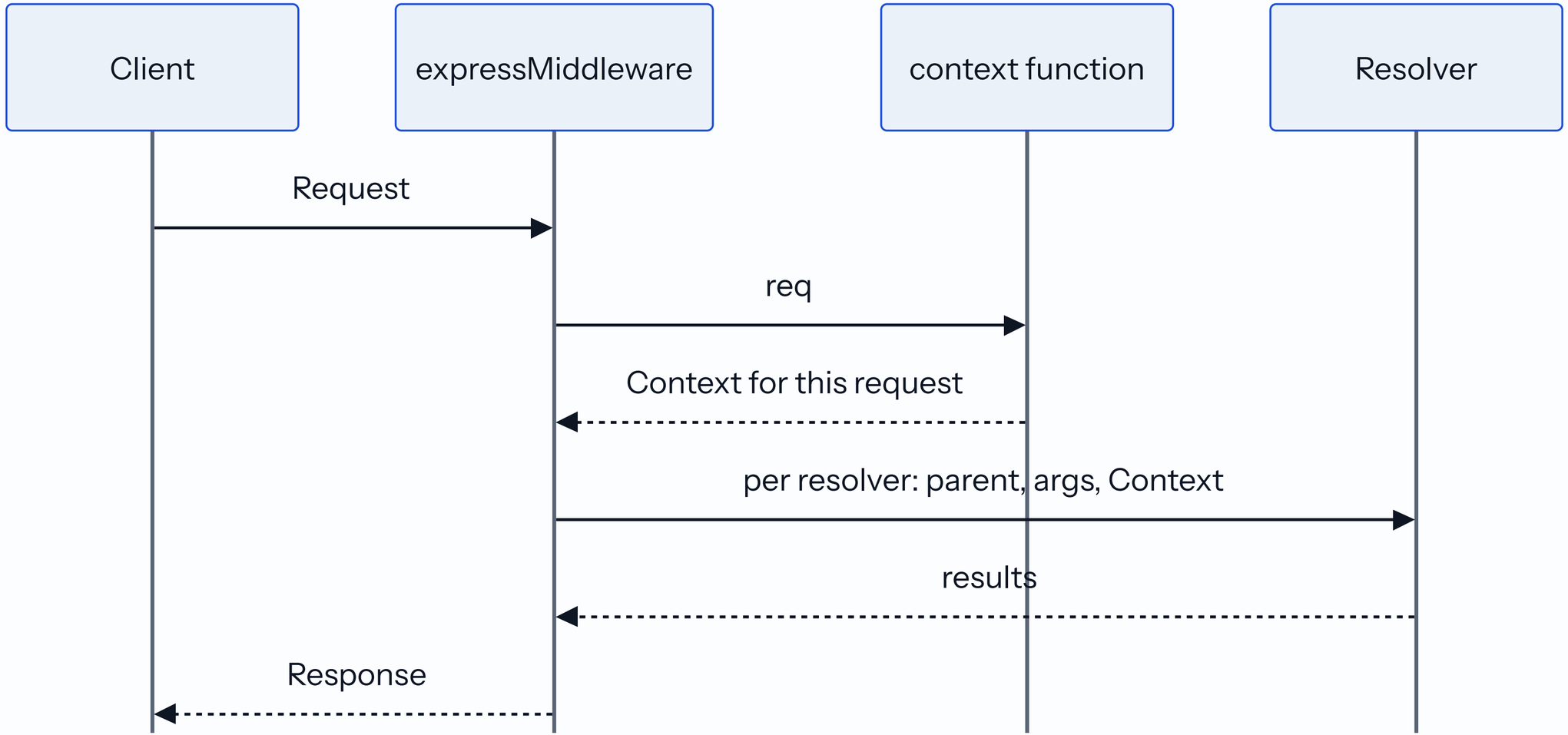
Exercise: errors as data

- Create PlaceOrderResult as a union
- Query the result with Type Conditions
- Mask unexpected errors with formatError

Design and structure

One context per request, and no one writes to it · Resolvers translate, business logic decides · One folder per domain, one schema for all

Where does a resolver get its dependencies?



The context function runs once per request

```ts
export type Context = Readonly<{
  requestId: string
  loaders: Loaders
}>

export const createContext = async ({ req }: { req: Request }): Promise<Context> =>
  Object.freeze({
    requestId: req.get('x-request-id') ?? randomUUID(),
    loaders: createLoaders(),
  })

const server = new ApolloServer<Context>({ typeDefs, resolvers })
app.use('/graphql', express.json(), expressMiddleware(server, { context: createContext }))
```

What belongs in the context?

- Values for this request: ID, identity, loaders
- Shared services: pool and logger, created once at startup
- Intermediate results: never inside, resolvers share them through loaders

Why must no resolver write into it?

```
// Wrong: orderCount runs in parallel and sees nothing
orders: async (_parent, _args, context) => {
  context.orders = await db.orders.findMany()
  return context.orders
},
orderCount: (_parent, _args, context) => context.orders?.length ?? 0,

// Correct: both read the same Loader
orders: (_parent, _args, { loaders }) => loaders.orders.load('all'),
orderCount: async (_parent, _args, { loaders }) =>
  (await loaders.orders.load('all')).length,
```

Test resolvers without starting HTTP

```
const contextValue: Context = {
  requestId: 'test-1',
  loaders: createLoaders(),
}

const response = await createServer().executeOperation(
  { query: 'query RequestId { requestId }' },
  { contextValue },
)

assert(response.body.kind === 'single')
expect(response.body.singleResult.data).toEqual({ requestId: 'test-1' })
```

What makes the context unreliable

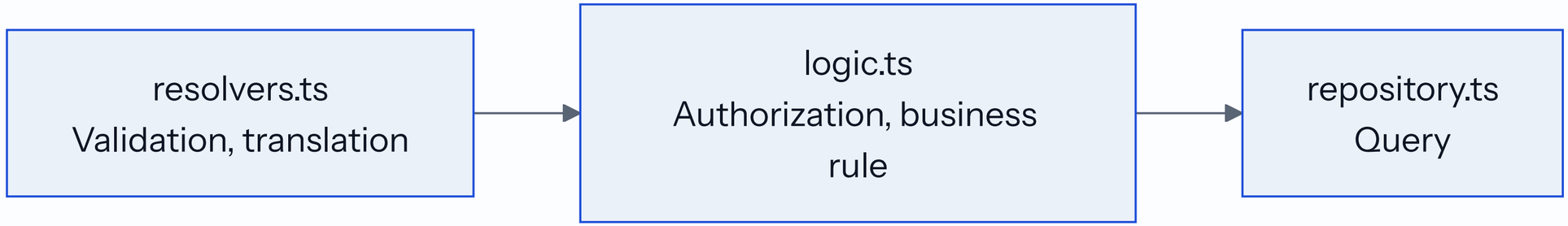
- Loaders in the module: the cache outlives the request
- Heavy work: every request pays, even without need
- Raw error: Apollo responds with status 500

What does this resolver do?

```
removeOrderItem: async (_parent, { input }, { viewer }) => {
  if (!viewer) throw unauthenticated()                    // Identity
  const order = await db.orders.findById(input.orderId)   // Query
  if (!order || order.customerId !== viewer.id)           // Authorization
    return orderNotFound(input.orderId)
  if (order.status !== 'DRAFT') return orderNotEditable(order) // Business rule
  const items = order.items.filter(/* ... */)
  const saved = await db.orders.saveItems(order.id, items) // Query
  return { __typename: 'ChangeOrderItemsSuccess', order: saved } // Translation
},
```

What a check needs defines its layer

- Check only needs the input: resolver
- Decision needs identity or status: business logic
- Access needs SQL: Data Access



Business logic knows no GraphQL

```ts
// orders/logic.ts
export const canChangeItems = (order: OrderRow) => order.status === 'DRAFT'

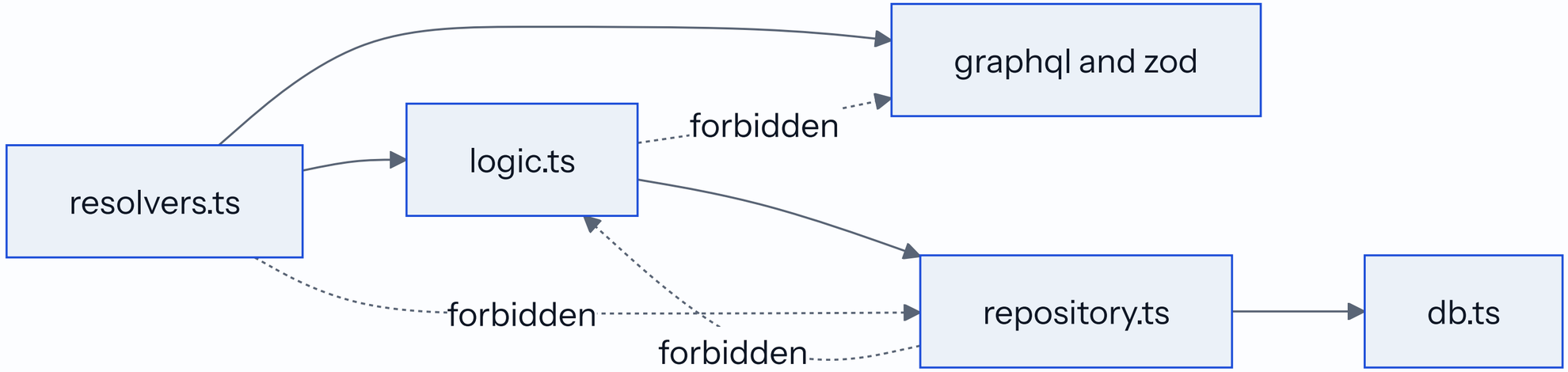
export const removeOrderItem = async (viewer: Viewer, input: RemoveInput) => {
  const order = await findOrder(input.orderId)
  if (!order || order.customerId !== viewer.id)
    return { ok: false, reason: 'NOT_FOUND', orderId: input.orderId }
  if (!canChangeItems(order)) return { ok: false, reason: 'NOT_EDITABLE', order }
  const items = order.items.filter((item) => item.productId !== input.productId)
  return { ok: true, order: await saveOrderItems(order.id, items) }
}
```

The resolver translates in both directions

```
// orders/resolvers.ts
const toGraphQL = (result: ChangeItemsResult) => {
  if (result.ok) return { __typename: 'ChangeOrderItemsSuccess', order: result.order }
  if (result.reason === 'NOT_FOUND') return orderNotFound(result.orderId)
  return orderNotEditable(result.order)
}

removeOrderItem: async (_parent, { input }, context) =>
  toGraphQL(
    await removeOrderItem(requireViewer(context), parseInput(RemoveOrderItemInput, input)),
  ),
```

Imports point only downward

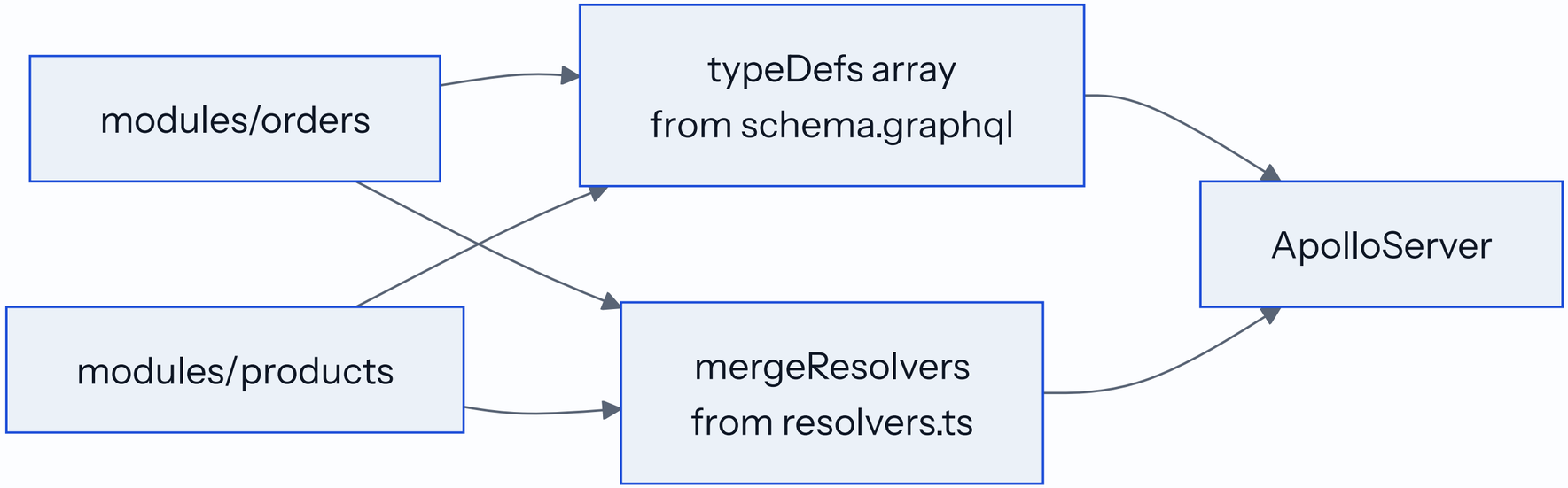


What glues the layers back together

- Shortcut: the resolver calls the repository directly
- Leakage: GraphQLError or __typename appear in business logic
- Moving down: the business rule becomes a WHERE clause in the repository

One folder per domain

- Each module contains schema.graphql, resolvers.ts, logic.ts, repository.ts
- The layers stay, they are just grouped per domain
- A module exposes only what its index.ts exports



Each module extends Query and Mutation

```graphql
# Root: empty, just so there is something to extend
type Query
type Mutation

# modules/orders/schema.graphql
extend type Query {
  orders: [Order!]!
}
extend type Mutation {
  removeOrderItem(input: RemoveOrderItemInput!): ChangeOrderItemsResult!
}

# modules/products/schema.graphql
extend type Query {
  products: [Product!]!
}
```

Assembly is your own code

```
// modules/orders/index.ts
export const typeDefs = readFileSync(
  new URL('./schema.graphql', import.meta.url), 'utf8',
)
export { resolvers } from './resolvers.ts'

// schema.ts
const modules = [orders, products]

export const typeDefs = [rootTypeDefs, ...modules.map((m) => m.typeDefs)]
export const resolvers = mergeResolvers(...modules.map((m) => m.resolvers))
```

Two modules, one field

```
export const mergeResolvers = (...modules: ResolverMap[]) => {
  const merged: ResolverMap = {}
  for (const module of modules) {
    for (const [type, fields] of Object.entries(module)) {
      merged[type] ??= {}
      for (const [field, resolver] of Object.entries(fields)) {
        if (Object.hasOwn(merged[type], field))
          throw new Error(`Two modules define the resolver ${type}.${field}`)
        merged[type][field] = resolver
      }
    }
  }
  return merged
}
```

A test secures the merge

```
it('throws when two modules define the same Query field', () => {
  const orders = { Query: { catalog: () => 'from orders' } }
  const products = { Query: { catalog: () => 'from products' } }

  expect(() => mergeResolvers(orders, products)).toThrow(
    'Two modules define the resolver Query.catalog',
  )
})
```

Schema-first or Code-first?

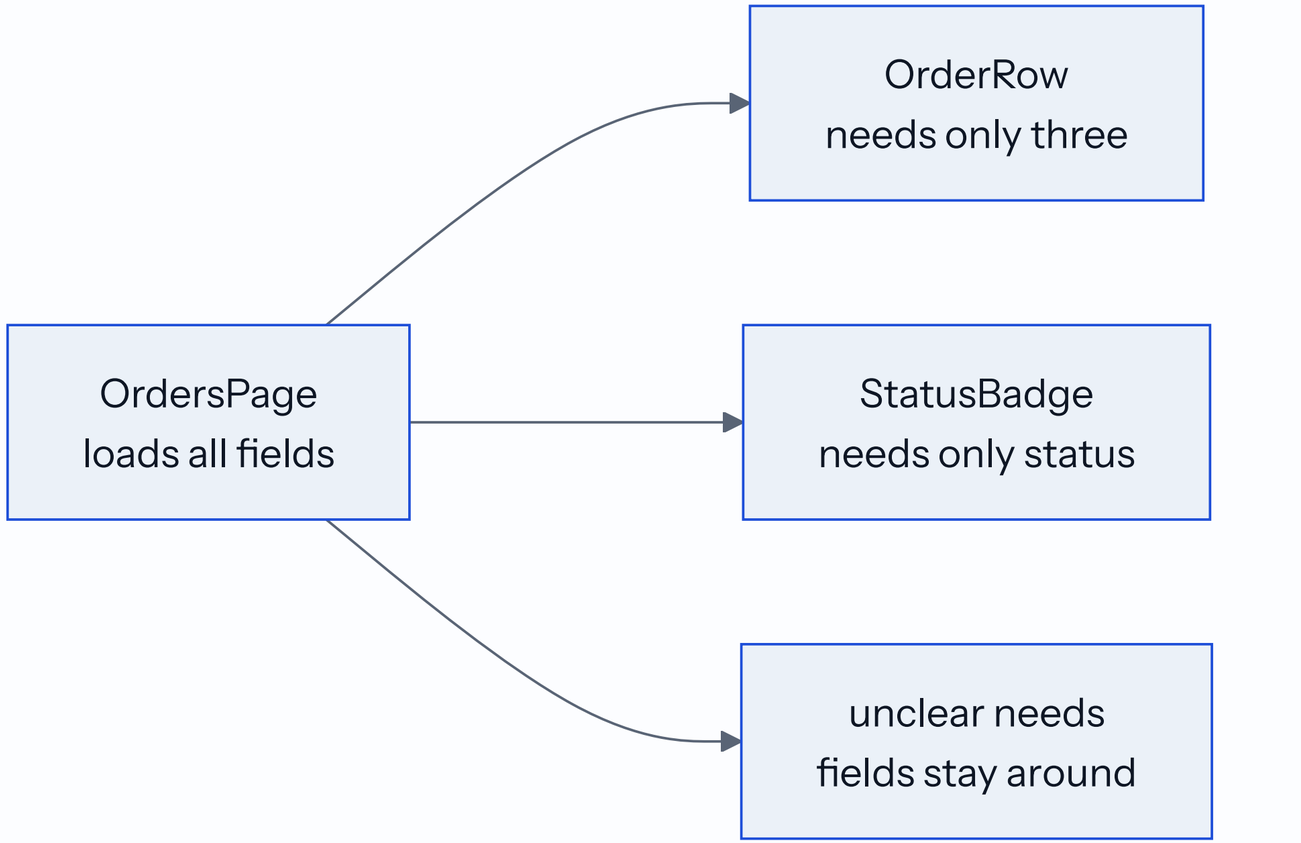
- Schema-first: SDL is the source, and resolvers follow it
- Code-first: TypeScript is the source, and SDL is generated
- Workshop standard: Schema-first, with Code-first as an alternative

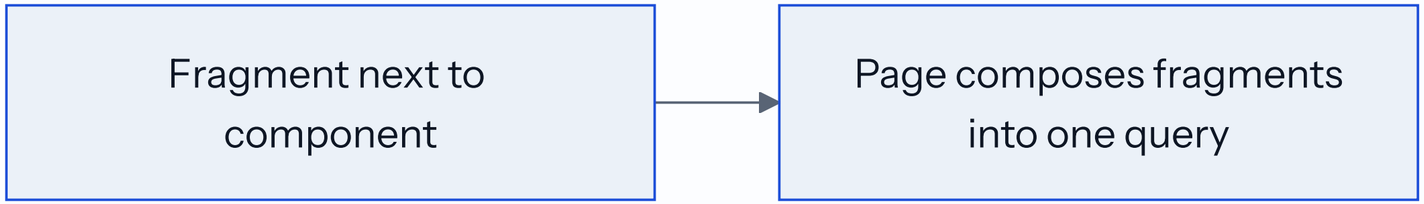
```
// Code-first with graphql, without an SDL file
const Product = new GraphQLObjectType({
  name: 'Product',
  fields: { id: { type: new GraphQLNonNull(GraphQLID) } },
})
```

Apollo Client

GraphQL in React with Apollo Client · One object, one client cache entry · Pagination with connections

One query for everything





Set up Apollo Client

- ApolloClient connects HttpLink and cache
- ApolloProvider makes it available to all components
- Hooks like useQuery read and load data

```
import { ApolloClient, HttpLink, InMemoryCache } from '@apollo/client'

export const client = new ApolloClient({
  link: new HttpLink({ uri: 'http://localhost:4000/graphql' }),
  cache: new InMemoryCache(),
})
```

Fragments next to the component

```
import { gql, type TypedDocumentNode } from '@apollo/client'
import { useFragment } from '@apollo/client/react'

type OrderRowData = { __typename: 'Order'; id: string; status: string; totalCents: number }

export const ORDER_ROW: TypedDocumentNode<OrderRowData> = gql`
  fragment OrderRow on Order { id status totalCents }
`

export const OrderRow = ({ order }: { order: { __typename: 'Order'; id: string } }) => {
  const { data, complete } = useFragment({ fragment: ORDER_ROW, from: order })
  return complete ? <li>{data.id}: {data.status}</li> : null
}
```

The page composes

```
import { useQuery } from '@apollo/client/react'

const ORDERS_PAGE: TypedDocumentNode<OrdersPageData, OrdersPageVariables> = gql`
  query OrdersPage($status: OrderStatus) {
    orders(status: $status) { id ...OrderRow }
  }
  ${ORDER_ROW}
`

export const OrdersPage = () => {
  const { data, loading, error } = useQuery(ORDERS_PAGE, { variables: { status: 'PLACED' } })
  if (loading) return <p>Lädt …</p>
  if (error) return <p role="alert">{error.message}</p>
  return <ul>{data?.orders.map((o) => <OrderRow key={o.id} order={o} />)}</ul>
}
```

What data components usually get wrong

- Loading state and error state are not rendered
- Values are built into the query as strings
- One query per component creates a waterfall

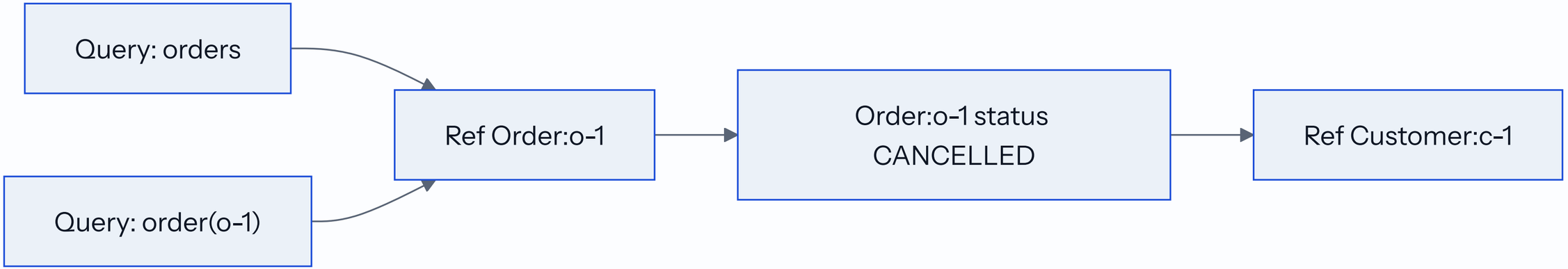
Exercise: Order overview in React

- Set up the client and provider
- Create one fragment per row
- Render loading, error, and empty list

Same data, two places

- List: status PLACED
- Detail after cancellation: status CANCELLED
- Without normalization: two truths

Normalization



A mutation updates the cache

```
const CANCEL_ORDER: TypedDocumentNode<CancelOrderData, CancelOrderVariables> = gql`
  mutation CancelOrder($input: CancelOrderInput!) {
    cancelOrder(input: $input) {
      order { id status }
    }
  }
`


const [cancelOrder, { loading }] = useMutation(CANCEL_ORDER)
```

Feel faster

```
cancelOrder({
  variables: { input: { orderId } },
  optimisticResponse: {
    cancelOrder: {
      order: { __typename: 'Order', id: orderId, status: 'CANCELLED' },
    },
  },
})
```

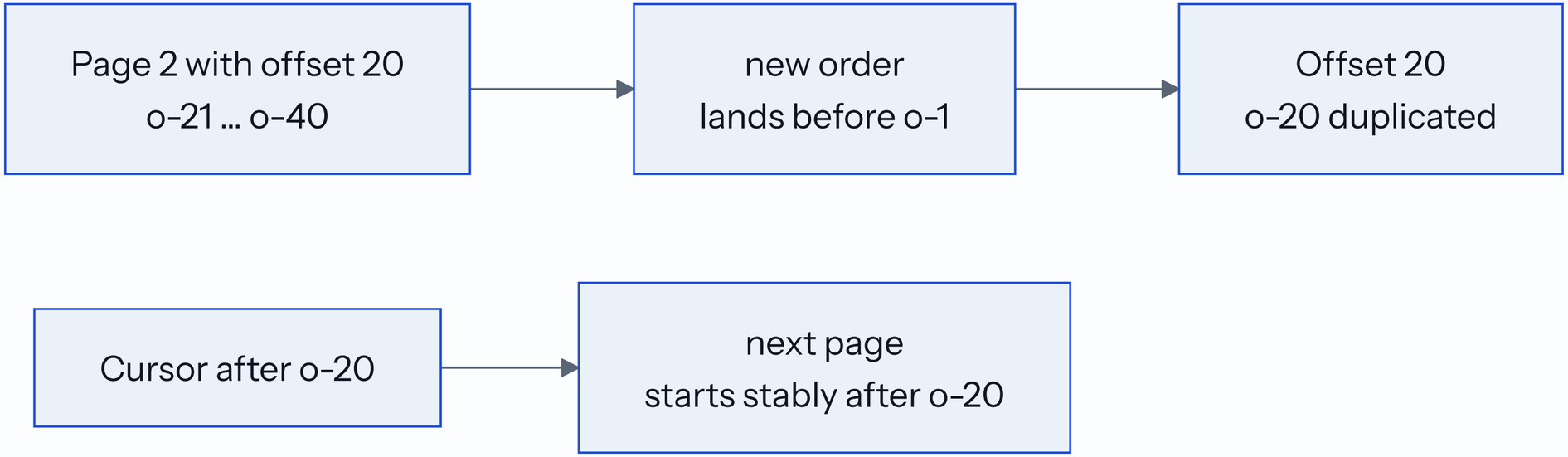
When the client cache shows wrong data

- Objects without an id are not merged
- Deleted entries remain in lists
- The cache is misused as global app state

Exercise: cancel optimistically

- Mutation CancelOrder with id and status
- Add __typename to optimisticResponse
- Check list and detail without a refetch

Lists without limits



The connection pattern

```
type Query {
  orders(first: Int = 20, after: String): OrderConnection!
}

type OrderConnection {
  edges: [OrderEdge!]!
  pageInfo: PageInfo!
}

type OrderEdge {
  node: Order!
  cursor: String!
}

type PageInfo {
  hasNextPage: Boolean!
  endCursor: String
}
```

The resolver

```
const encode = (id: string) => Buffer.from(id).toString('base64url')
const decode = (cursor: string) => Buffer.from(cursor, 'base64url').toString()

const orders = async (_: unknown, { first, after }: { first: number; after?: string | null }) => {
  const limit = Math.min(first, 100)
  const rows = await db.orders.after(after ? decode(after) : null, limit + 1)
  const page = rows.slice(0, limit)
  const last = page.at(-1)
  return {
    edges: page.map((node) => ({ node, cursor: encode(node.id) })),
    pageInfo: { hasNextPage: rows.length > limit, endCursor: last && encode(last.id) },
  }
}
```

Paging with Apollo Client

```
const cache = new InMemoryCache({
  typePolicies: { Query: { fields: { orders: relayStylePagination() } } },
})

const { data, fetchMore } = useQuery(ORDERS)
const { pageInfo } = data.orders

pageInfo.hasNextPage && (
  <button onClick={() => fetchMore({ variables: { after: pageInfo.endCursor } })}>
    Mehr laden
  </button>
)
```

When pagination becomes wrong or expensive

- Cursor on non-unique sorting
- totalCount is expensive on large tables
- Connections for lists with three entries

Exercise: page through orders

- Convert Query.orders to a connection
- Determine hasNextPage with one extra entry
- Load more in the client with fetchMore

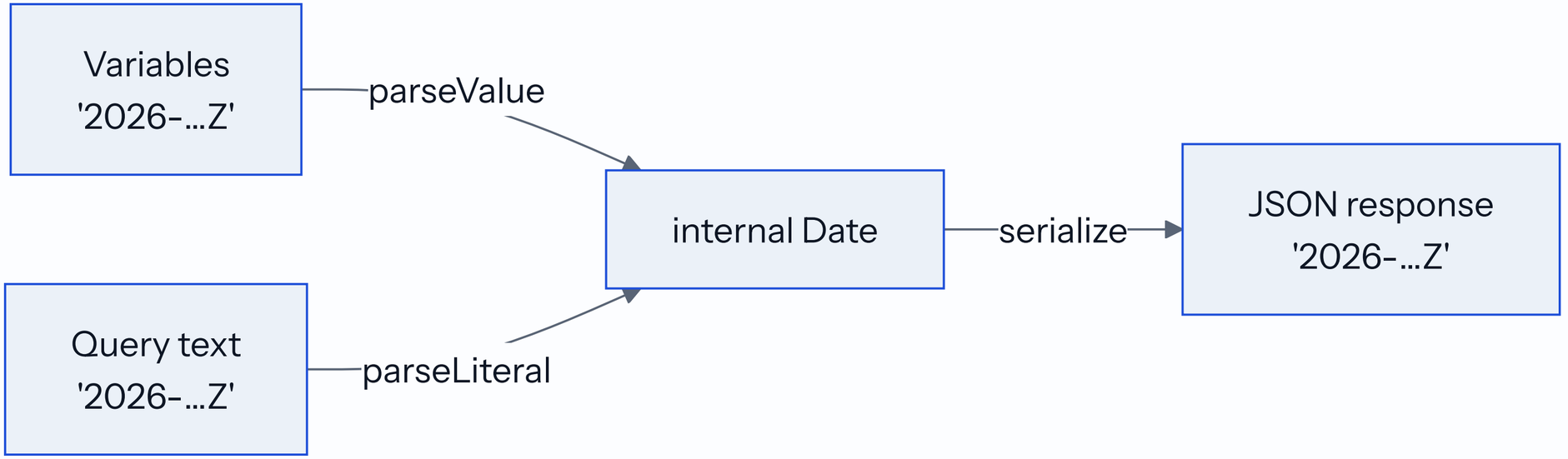
Scalars, testing, REST

Custom scalars validate values at the boundary · Test operations the way clients send them · Integrating existing REST APIs

String is not a date

- createdAt as String: which format?
- email as String: any text is valid
- Validation: scattered across every resolver

Three directions of a scalar



Scalar in the schema

```
scalar DateTime @specifiedBy(url: "https://www.rfc-editor.org/rfc/rfc3339")
scalar EmailAddress

type Order {
  id: ID!
  createdAt: DateTime!
  contactEmail: EmailAddress
}

type Query {
  orders(createdAfter: DateTime): [Order!]!
}
```

Implement DateTime

```ts
import { GraphQLError, GraphQLScalarType, Kind } from 'graphql'

const RFC3339 = /^\d{4}-\d\d-\d\dT\d\d:\d\d:\d\d(\.\d+)?(Z|[+-]\d\d:\d\d)$/

const toDate = (value: unknown) => {
  if (typeof value !== 'string' || !RFC3339.test(value))
    throw new GraphQLError('DateTime erwartet RFC 3339 mit Offset')
  return new Date(value)
}

export const DateTime = new GraphQLScalarType<Date, string>({
  name: 'DateTime',
  serialize: (value) => (value as Date).toISOString(),
  parseValue: toDate,
  parseLiteral: (ast) => toDate(ast.kind === Kind.STRING ? ast.value : null),
})
```

Do not build it yourself

- graphql-scalars provides DateTime, EmailAddress, URL, and more
- Resolvers register scalars under their names
- Domain rules stay in the domain logic

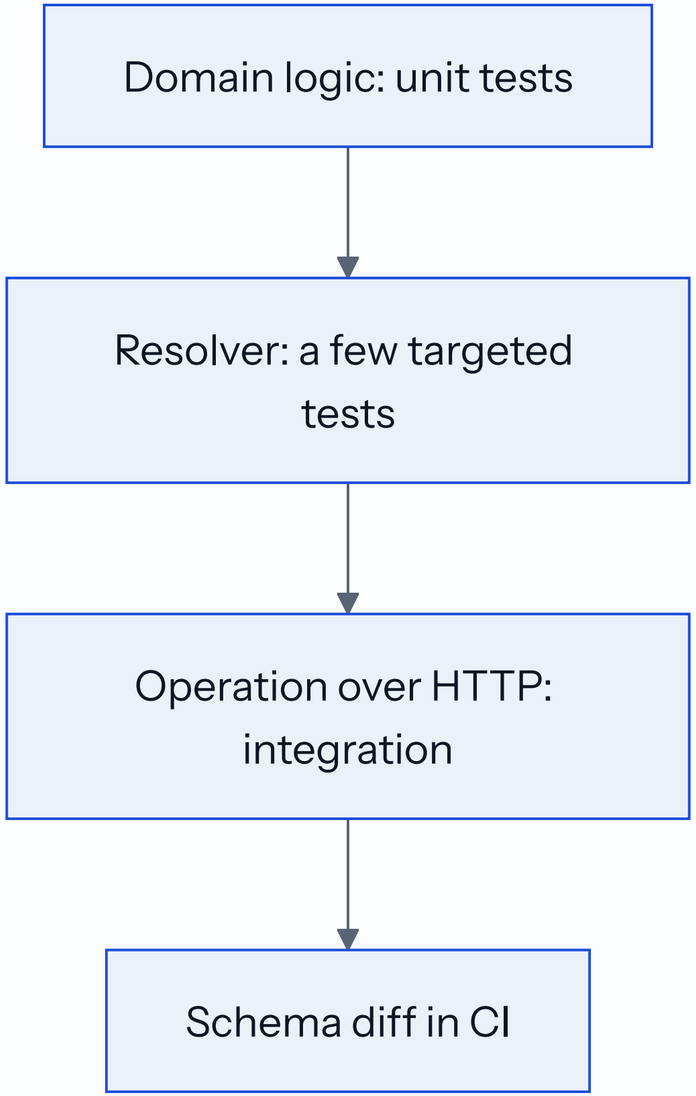
```
import { DateTimeResolver, EmailAddressResolver } from 'graphql-scalars'

export const resolvers = {
  DateTime: DateTimeResolver,
  EmailAddress: EmailAddressResolver,
  Query: { orders: listOrders },
}
```

Exercise: createdAt as DateTime

- Declare DateTime in the schema
- Register the scalar in the resolvers
- Reject an invalid date by test

What should you test?



Domain logic first

```
import { expect, it } from 'vitest'
import { placeOrder } from './orders.ts'

it('meldet OutOfStock für leeres Lager', async () => {
  const result = await placeOrder('c-1', {
    items: [{ productId: 'p-2', quantity: 1 }],
  })
  expect(result.__typename).toBe('OutOfStock')
})
```

Testable app

```
export const createApp = async () => {
  const server = new ApolloServer<Context>({ typeDefs, resolvers })
  await server.start()
  const app = express()
  app.use('/graphql', express.json(), expressMiddleware(server, {
    context: async ({ req }) => ({
      customerId: await customerFromToken(req.headers.authorization),
      loaders: createLoaders(),
    }),
  }))
  return app
}
```

Operation over HTTP

```
const post = async (query: string, variables = {}, headers: Record<string, string> = {}) => {
  const response = await fetch(url, {
    method: 'POST',
    headers: { 'content-type': 'application/json', ...headers },
    body: JSON.stringify({ query, variables }),
  })
  return { status: response.status, body: await response.json() }
}

it('liefert die Produkte', async () => {
  const { status, body } = await post('query Products { products { id name priceCents } }')
  expect(status).toBe(200)
  expect(body.errors).toBeUndefined()
})
```

Which tests check too little or too much

- Snapshot of the whole response: breaks on every field
- Only the happy path: error cases untested
- No schema diff: breaking changes only show up in production

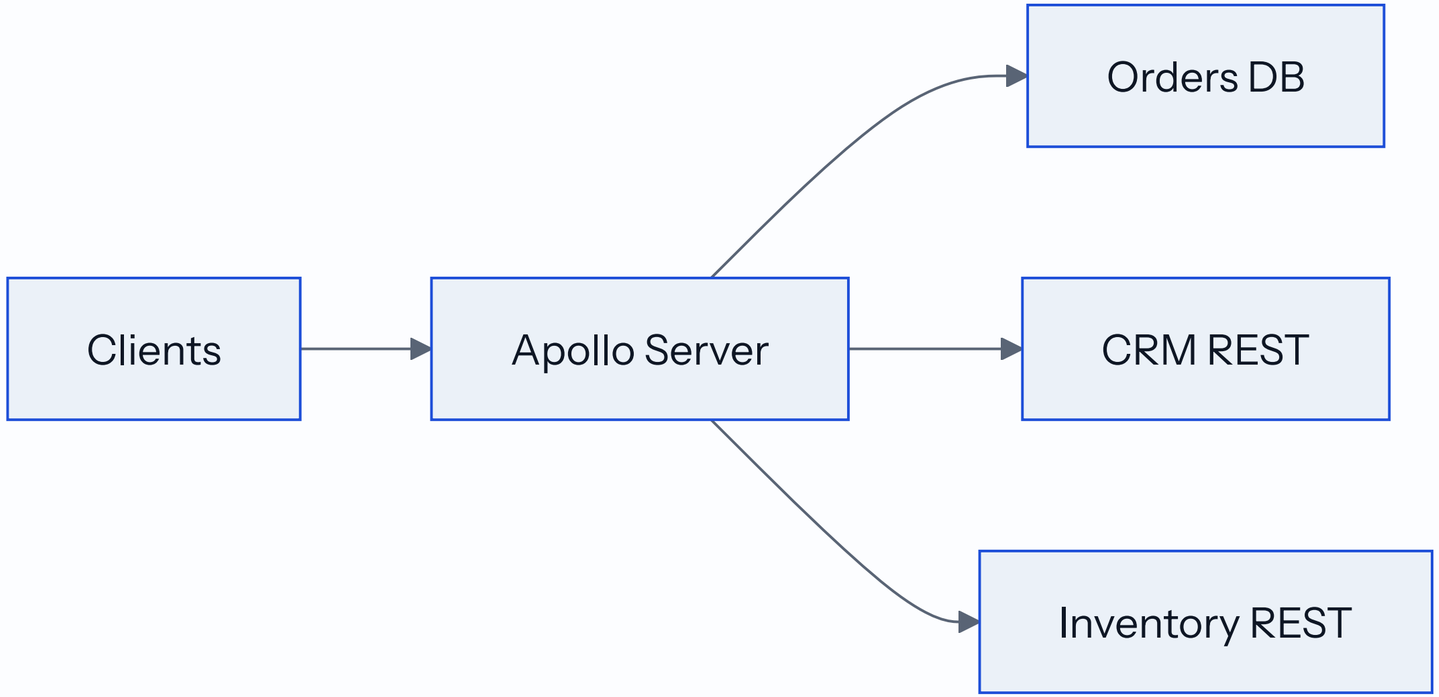
Exercise: tests for the shop

- Query products over HTTP
- placeOrder without a token: UNAUTHENTICATED
- Invalid variable: status 400

The reality

- Customers are in the CRM, inventory is in the inventory service
- Each service has its own REST API
- Clients want a single view

GraphQL as a facade



One client module per service

```
const fetchJson = async <T>(url: string): Promise<T> => {
  const response = await fetch(url, { signal: AbortSignal.timeout(1000) })
    .catch((error: Error) => {
      throw new GraphQLError('Lagerdienst nicht erreichbar', {
        extensions: { code: 'UPSTREAM_UNAVAILABLE' }, originalError: error,
      })
    })
  if (!response.ok) throw new GraphQLError(`Lagerdienst antwortet ${response.status}`, {
    extensions: { code: 'UPSTREAM_ERROR' },
  })
  return (await response.json()) as T
}
```

Resolver with loader

```
const createLoaders = () => ({
  stock: new DataLoader(async (ids: readonly string[]) => {
    const query = new URLSearchParams({ ids: ids.join(',') })
    const rows = await fetchJson<StockRow[]>(`${INVENTORY_URL}/stock?${query}`)
    const byId = new Map(rows.map((row) => [row.productId, row.available]))
    return ids.map((id) => byId.get(id) ?? null)
  }),
})

const resolvers = {
  Product: {
    available: (product: ProductRow, _args: unknown, context: Context) =>
      context.loaders.stock.load(product.id),
  },
}
```

What goes wrong when integrating REST

- Schema generated one-to-one from OpenAPI
- Service B waits for service A: waterfall
- Error messages from services forwarded to clients

Exercise: inventory from the inventory service

- Write a client module with timeout
- Batch calls with a loader
- Simulate a service outage

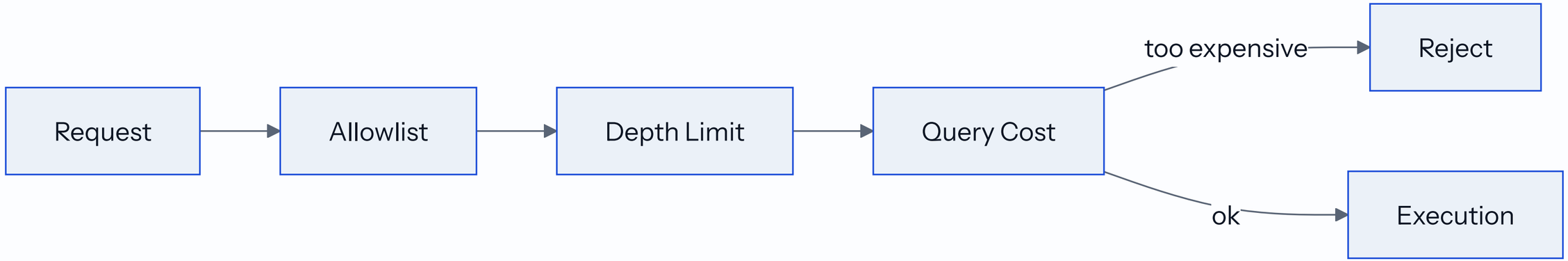
Limits, caching, schema evolution

How do you protect an open graph? · Caching at every level · Schema evolution without breaking changes

A query, heavy load

```
query {
  orders(first: 100) {
    edges { node {
      customer { orders(first: 100) {
        edges { node {
          customer { orders(first: 100) { edges { node { id } } } }
        } }
      } }
    } }
  }
}
```

Three protection layers



Depth limit as a validation rule

```
import { GraphQLError, Kind, type ValidationRule } from 'graphql'

const depthLimit = (max: number): ValidationRule => (context) => ({
  Field(node, _key, _parent, _path, ancestors) {
    const depth = ancestors.filter((a) => 'kind' in a && a.kind === Kind.FIELD).length + 1
    if (depth <= max) return
    context.reportError(new GraphQLError(`Depth ${depth} über ${max}`, { nodes: node }))
    return false
  },
})

new ApolloServer({ typeDefs, resolvers, validationRules: [depthLimit(6)] })
```

Query cost with variables

```
const costLimit = (maxCost: number): ApolloServerPlugin => ({
  async requestDidStart() {
    return {
      async didResolveOperation({ operation, request }) {
        const cost = estimateCost(operation?.selectionSet, request.variables ?? {})
        if (cost > maxCost)
          throw new GraphQLError(`Query Cost ${cost} über ${maxCost}`, {
            extensions: { code: 'QUERY_TOO_EXPENSIVE', http: { status: 400 } },
          })
      },
    }
  },
})
```

Persisted operations as an allowlist

```
const operations = new Map(manifest.operations.map(({ body }) => [sha256(body), body]))

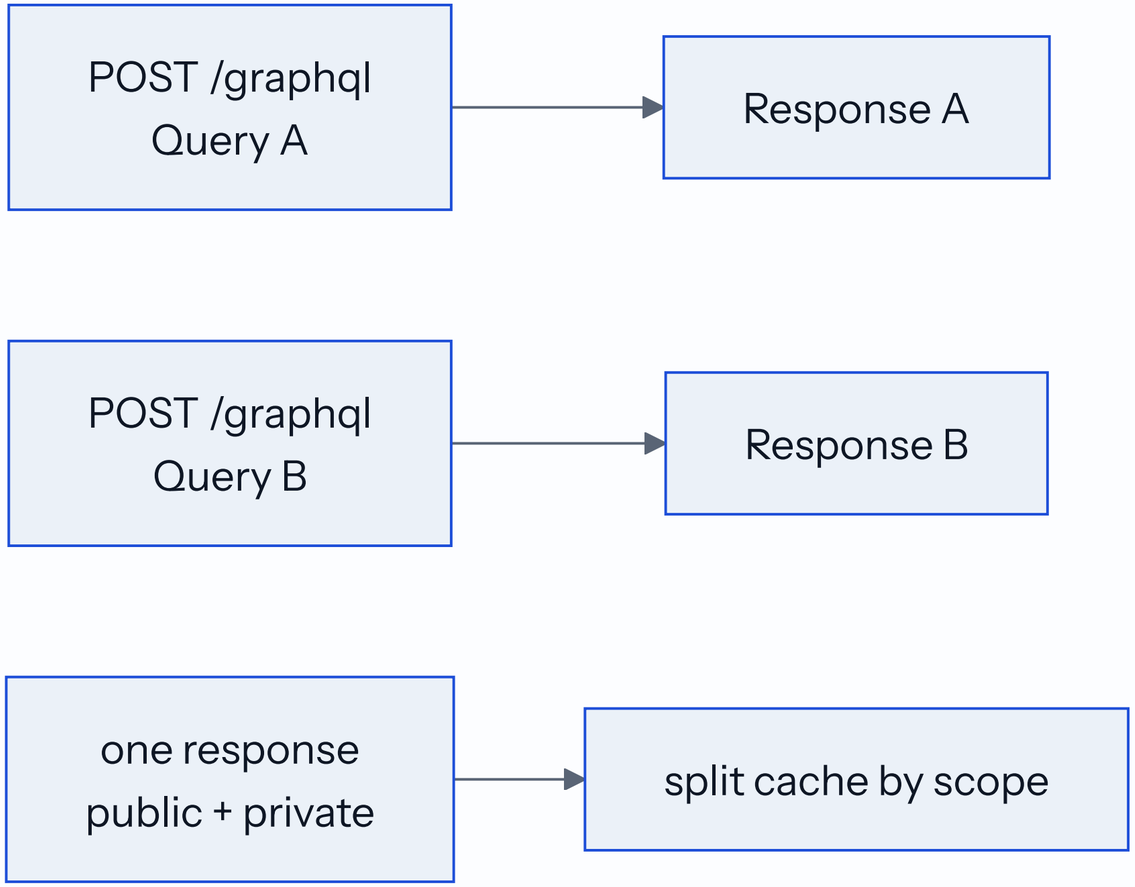
const allowlist: express.RequestHandler = (request, response, next) => {
  const hash = request.body?.extensions?.persistedQuery?.sha256Hash
  const query = typeof hash === 'string' ? operations.get(hash) : undefined
  if (!query) {
    response.status(400).json({ errors: [{ message: 'Unbekannte Operation' }] })
    return
  }
  request.body = { ...request.body, query, extensions: {} }
  next()
}

app.use('/graphql', express.json(), allowlist, expressMiddleware(server))
```

Which protections only seem to protect

- Rate limit per request instead of per query cost
- Introspection turned off and treated as secure
- Manifest and deployed client are out of sync

Why no HTTP cache?



The levels



Cache hints in the schema

```
enum CacheControlScope { PUBLIC PRIVATE }
directive @cacheControl(maxAge: Int, scope: CacheControlScope, inheritMaxAge: Boolean)
  on FIELD_DEFINITION | OBJECT | INTERFACE | UNION

type Product @cacheControl(maxAge: 60) {
  id: ID!
  name: String!
  priceCents: Int!
  stock: Int! @cacheControl(maxAge: 5)
}

type Order @cacheControl(maxAge: 10, scope: PRIVATE) {
  id: ID!
  status: OrderStatus!
}
```

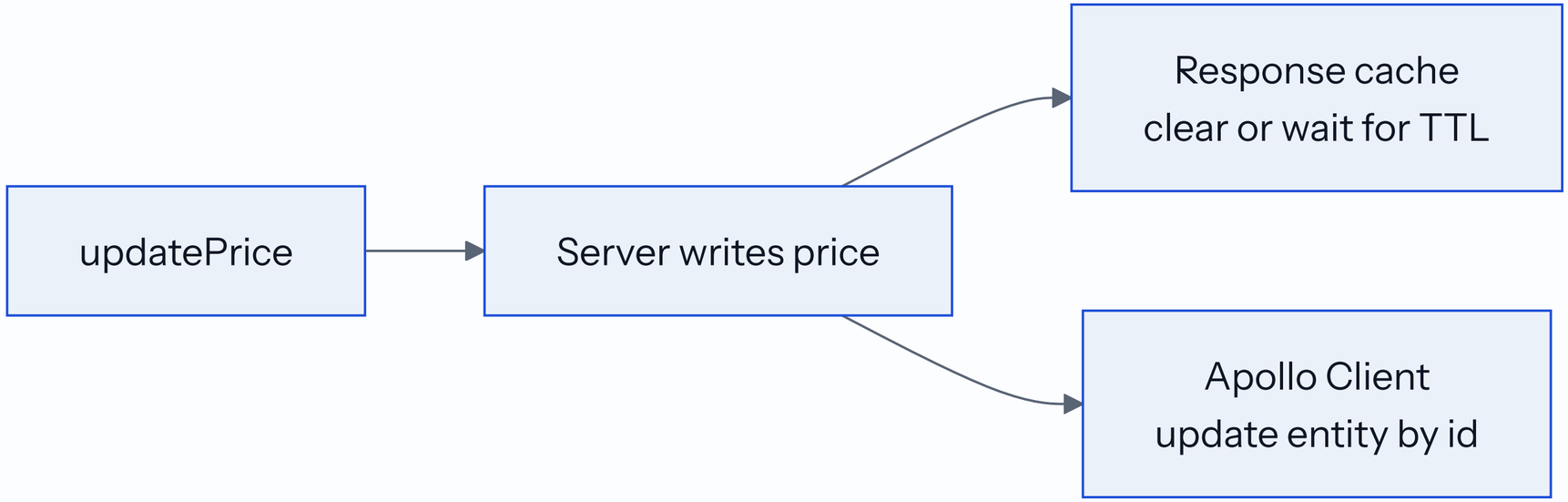
Response cache in Apollo Server

```
import responseCachePlugin from '@apollo/server-plugin-response-cache'
import { InMemoryLRUCache } from '@apollo/utils.keyvaluecache'

export const responseCache = new InMemoryLRUCache<string>()

const server = new ApolloServer({
  typeDefs,
  resolvers,
  plugins: [
    responseCachePlugin({
      cache: responseCache,
      sessionId: async ({ request }) =>
        request.http?.headers.get('authorization') ?? null,
    }),
  ],
})
```

Cache invalidation



Wrong caching is worse than none

- Private data cached at the CDN
- DataLoader as a replacement for a cache
- Caching everywhere, but no metrics

No v2 in the path

- One graph serves many client versions
- Apps in the store keep running for years
- Clients request only what they need

Expand and contract



Deprecation in the schema

```
type Order {
  id: ID!
  total: Int! @deprecated(reason: "Nutze totalPrice mit Währung.")
  totalPrice: Money!
}

type Money {
  amountCents: Int!
  currency: String!
}
```

Detect breaking changes

- Field or type removed: breaking
- New non-null argument or input field without default: breaking
- New enum value: risky for old clients

What makes schema changes risky

- Field renamed because the name is nicer
- Field deprecated but never removed
- Field usage cannot be assigned without a client id

Exercise: from total to totalPrice

- Add totalPrice, mark total as deprecated
- Run schema diff in CI
- Measure usage of the old field

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen