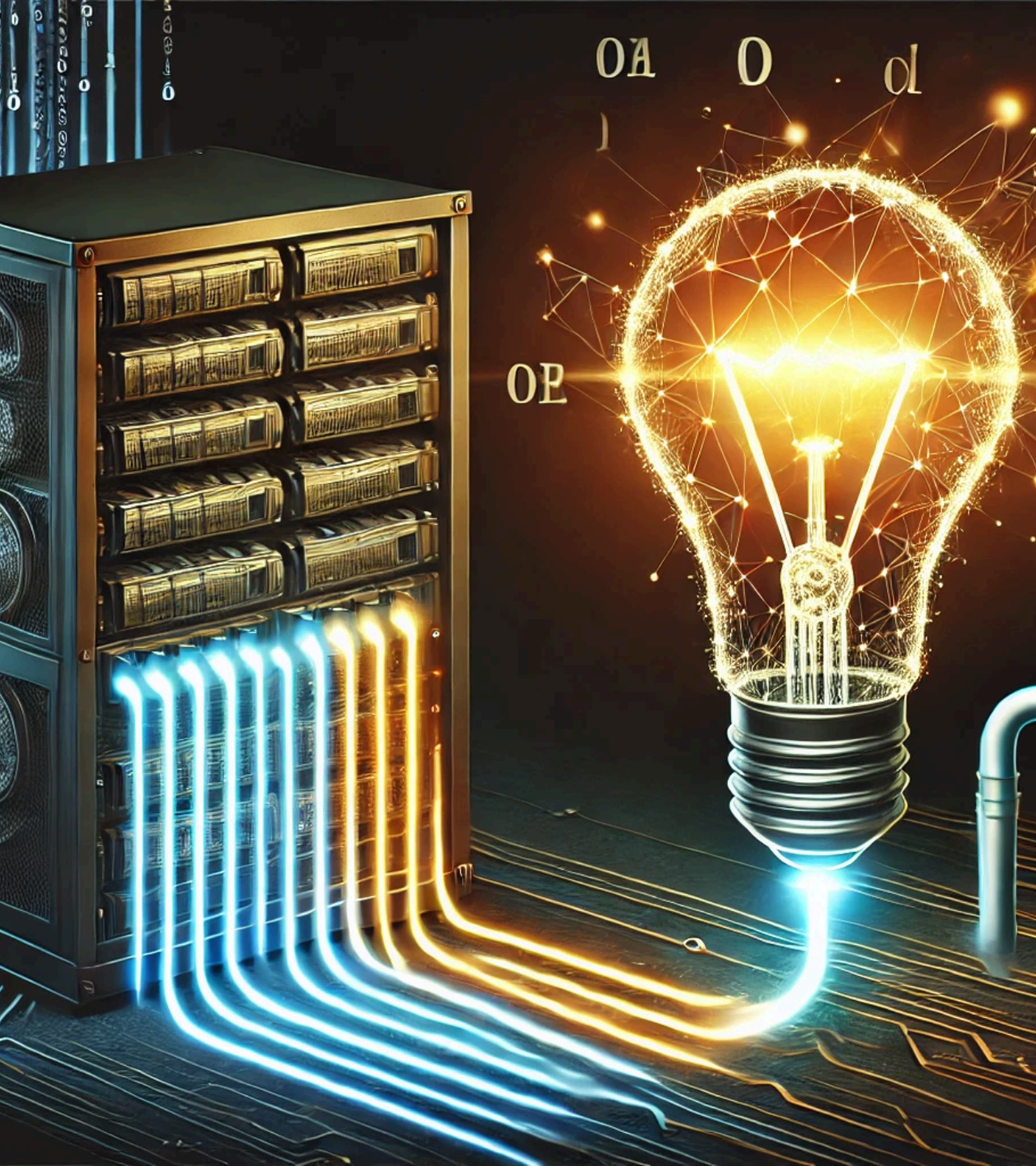




Software Architecture

Week 4a · CIS 3990 / 2500

Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

Today's Big Question

What makes software easy to change, and how do you coordinate change safely across contributors and agents?

The answer is similar whether we are talking about agents or humans!

- **Architectural patterns** help reduce complexity globally
- Careful **module scoping** keeps it easy to get pieces of the software working together

Why Does Architecture Matter?

When a codebase grows beyond what one person can hold in working memory, **architecture becomes the tool for managing complexity.**

Let's chat:

What are the bad things that can happen when a system is poorly organized?

Why Architecture Matters

When a codebase grows beyond what one person can hold in working memory, **architecture becomes the tool for managing complexity**.

Thoughtful architecture enables:

- Reuse: shared implementations of common functionality.
- Abstraction: define building blocks from higher-level components and abstractions.
- Encapsulation of properties like security: We can "bake in" certain constraints and properties into our modules.
- Decomposition into independent units: scoping sub-components in a way that has focused knowledge, interactions, and changes.

When is Software "Good"?

Going beyond functionality ("does it work?"), what are some qualities of software that make it "good?"

Holistic Properties Must Be Designed In

Security, concurrency, consistency, and privacy cannot be bolted on after the fact — they must be designed into the architecture from the start.

- **Security** — authentication and authorization boundaries, data sanitization at system edges
- **Concurrency** — shared state minimized, race conditions designed out
- **Consistency** — invariants enforced at write points, not scattered across reads
- **Privacy** — sensitive data isolated, access logged

Agents usually produce plausible solutions, so the architecture must make violations visible as loud type errors or test failures.

Example: Security & Privacy

A web-app with access to private data must:

1. Receive HTTP requests and parse out data;
2. **Authenticate** the user who sent the request, verifying that they are who they say;
3. **Authorize** the request based on the permissions that we have for the user;
4. Pass the request on to the proper route and handle it there.

We can rely on each route handler to authenticate and authorize, or we can design the system to do this as a matter of course.

Example: Per-Route Auth ❌

Without a structural feature to enforce, each endpoint must remember to authenticate.

```
// Endpoint 1: Agent added code to check auth
app.post('/api/messages', (req, res) => {
  if (!req.headers.authorization) return res.status(401).send('Unauthorized');
  const user = validateToken(req.headers.authorization);
  if (!hasPermission(user, 'SEND_MESSAGES')) return res.status(403).send('Forbidden');

  return user.messages()
});

// Endpoint 2: Agent forgot – BUG!!!
app.delete('/api/messages/:id', (req, res) => {
  const message = await Message.findById(req.params.id);
  await message.delete(); // ⚠ Anyone can delete any message!
```

Example: Auth as a Layer

Here's an Express application where authentication and authorization are registered globally, applying to all routes automatically. *Don't worry about the details here.*

```
const app = express();

// Authentication middleware runs on every request
app.use(authenticate);    // Validates JWT, attaches user to req.user
app.use(authorize);       // Checks user permissions

// Now all routes below are protected automatically
app.post('/api/messages', createMessage);
app.delete('/api/messages/:id', deleteMessage);
app.patch('/api/guilds/:id', updateGuild);
```

The middleware architecture **makes it impossible to forget** authorization on a new

9

Let the System Work For You

The rules of our system can make the work of writing new code easier and less error-prone.

With finite context, an agent can't always take every global requirement into account—but a carefully designed system means *we don't care!*

It's much harder to waste a shot in bowling if you have the bumpers up!



Some Desirable Properties of a System

How to let the system work for you.

Desirable Properties of a System: Decomposability

Decomposability is the property that lets you understand and modify one part of the system without needing to understand all of it.

Techniques:

- **Interfaces** — contracts that hide implementation details
- **Services** — independently deployable units
- **Layers** — HTTP edge, application logic, domain model, persistence

Agents can modify one layer without needing full context on the others.

Decomposition in Spacebar

Spacebar is organized into four main layers, each with distinct responsibilities.

HTTP Edge (`src/api/routes/`, `src/api/middlewares/`) — parses requests, authenticates, returns responses

Application Logic (`src/util/util/Permissions.ts`, `emitEvent()`) — decides what a request is allowed to do, and who else needs to hear about it

Domain Model (`src/database/entities/`) — TypeORM entities: types with corresponding database tables

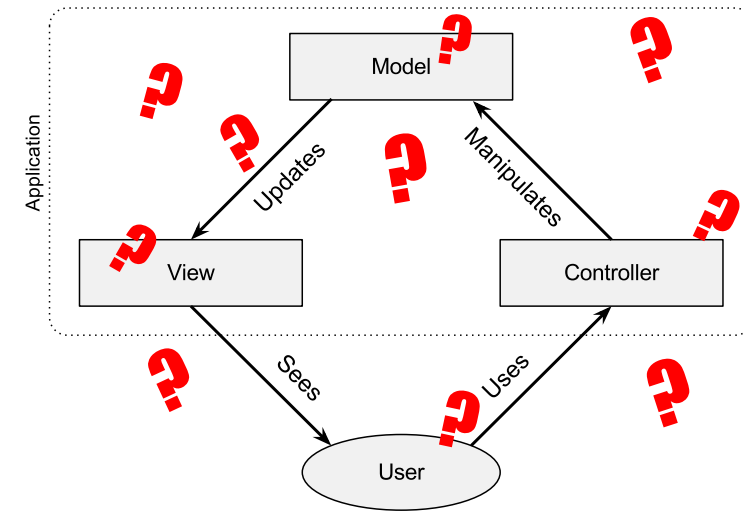
Persistence (PostgreSQL) — stores relational data, enforces foreign keys

You'll use the LLM to help figure out what advantages these layers provide and how they communicate.

Desirable Properties of a System: Consistent Use of Patterns & Styles

In a traditional software engineering course, you might learn about various architectural patterns & styles: *Model-View-Controller* or *Publisher-Subscriber* or *Monoliths* vs. *Microservices*...

- probably the agents will even build these things for you without your asking!
- probably you don't need to be able to recognize all of these patterns anymore!



In this course, we can look forward to working at a higher level of abstraction.



Patterns & Styles in Our Level of Abstraction

The agent will be responsible for making some design choices—with your help!—and then **documenting them!**

The documentation helps in the following ways:

- ADRs will be created by LLMs for LLMs to provide info-dense descriptions of a problem that don't crowd context
- Developer-facing documentation will contain more human-readable guides for how the system works
- `CLAUDE.md` reminds an agent of the locations of both kinds of documents
- Some agents can be used to enforce consistency whilst others work on new tasks (next slide)

Example: roborev



"roborev reviews every commit in the background and feeds findings back to your coding agent so the write -> review -> fix loop runs hands-off."

Source: [roborev](#)

17

roborev Makes Its Reviews

```
roborev v0.63.0
Workers: 0/4 | Closed: 0 | Open: 2 | ~$0.00 (0/2)

JobID ... .. Agent   Queued           Elapsed Status P/F Clo
-----
> 2      ... .. claude Jul 21 17:27      1m39s Done   F   no
  1      ... .. claude Jul 21 17:18      1m1s Done   F   no
```

Review #1 PennDraw (claude-code)

/Users/harrysmith/Documents/PennDraw 1c6f4ff on p5-st

Verdict: Fail [CLOSED]

Review Findings

Severity: Low

Location: penndraw/penndraw.py:96-104 (run)

Problem: The `animation` parameter of `run(animation=None)` is unused, and there's no validation preventing a user from registering `@pd.draw` while also manually calling `pd.advance()` in their own loop, or calling `pd.run()` without ever registering `@pd.setup/@pd.draw` alongside stray callback registrations left over from a prior script

Claude Is Asked to Fix

(Here I manually triggered this, but `roborev` can hook into a running agent to ask it to review, too.)

```
/roborev-fix

Searched for 1 pattern, read 1 file, listed 1 directory, ran 6 shell
commands

• Now I'll apply the fixes. Starting with the Low-severity docstring fix
on on_key_pressed/on_key_released (job 1, finding 2):

Read 1 file

• Update(penndraw/penndraw.py)
  | Added 8 lines
    897
    898
    899 def on_key_pressed(fn):
    900 +     """Regis Jump to bottom (click) ↓ hen a key is pre

>> accept edits on (shift+tab to cycle) · ← for agents
```

- Both original review jobs (1 and 2) are fixed, commented, and closed. Committed as c8e5c38:
 - Added `IndependentAxisScaleTests` in `tests/unittests.py` covering `set_x_scale/set_y_scale` in isolation plus a regression test for the `set_canvas_size()` axis-collapse bug.
 - Documented the Caps Lock/Shift behavior on `on_key_pressed/on_key_released`.
 - Fixed the inconsistent placeholder naming for `set_scale()` in `penndraw.md`.
 - Replaced `README.md`'s stale, drifting copy of the API reference with a pointer to `penndraw.md`.

The New Commit Gets Reviewed (& Passes)

roborevv0.63.0

Workers: 0/4 | Closed: 2 | Open: 1 | ~\$0.00 (0/3)

JobID	...	...	...	Agent	Queued	Elapsed	Status	P/F	Clo
3	...	...	...	claude	Jul 21 17:34	20s	Done	P	no
> 2	...	...	...	claude	Jul 21 17:27	1m39s	Done	F	yes
1	...	...	...	claude	Jul 21 17:18	1m1s	Done	F	yes

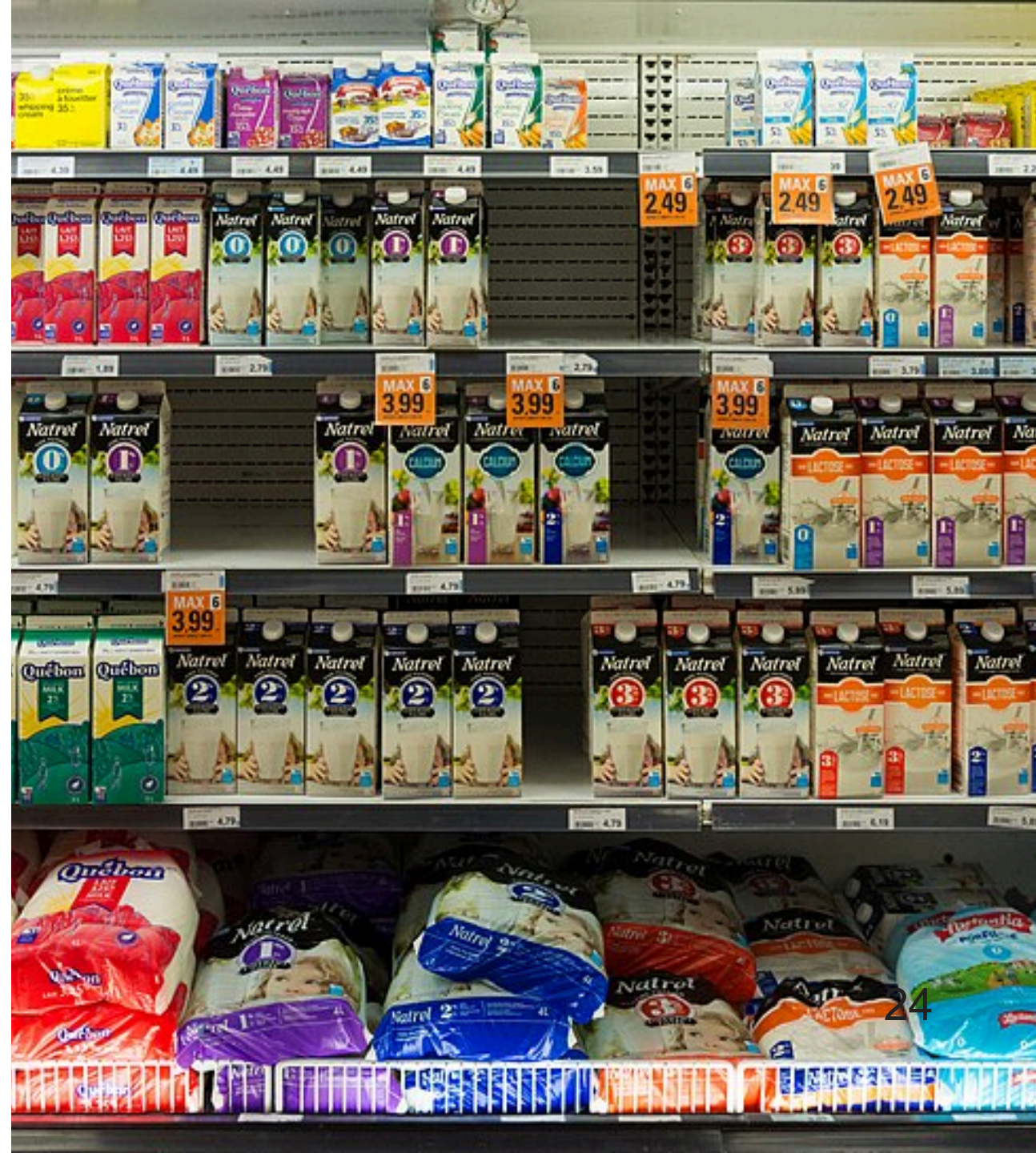
Demo: Features With and Without ADRs; Reviews

Desirable Properties of a System: Deep Modules

Deep modules (Ousterhout, [A Philosophy of Software Design](#)) have:

- **Small interfaces** — few public functions, clear contracts
- **Large implementations** — complex logic hidden inside

Like we saw last week with tool-calling, **confusion** can happen when there are too many options to consider.



Shallow vs. Deep

Shallow module (bad)

```
class PermissionChecker {  
    checkGuildPerms( ... )  
    checkChannelPerms( ... )  
    checkRolePerms( ... )  
    checkMemberPerms( ... )  
    checkOwnerPerms( ... )  
    computeFinal( ... )  
    ...  
}
```

Deep module (good)

```
class Permissions {  
    static finalPermission(  
        { user, guild, channel }  
    ): Permissions  
    ...  
}
```

We prefer to expose one method that produces the important final outcome rather than a collection of other methods that combine to do the same thing.

25

Why Agents Benefit from Deep Modules

A tangled codebase with ambiguous responsibilities produces agents that invent their own interfaces — hallucinated APIs that sound plausible but don't exist.

When an agent reads a deep module, there is:

- One entry point
- A clear input/output contract
- No irrelevant implementation detail

The agent can use the module correctly without reading the implementation.

Software at Scale: Databases & ORMs

"Here Be Dragons"

Old mapmakers used to put images of dragons and monsters at the edges of the map/globe to warn of the mysteries of the unknown.

Also probably because it looked cool.



Dragons, Closer Up



The Dragons: Databases & ORMs

In CIS 1200 and 1210, you learn about **data types** and **classes** in the context of a program.

- These define the sorts of things your software can represent and manipulate
- They are collections of other pieces of information, perhaps recursively.
- They live *only in the context of your program's memory*.

Spacebar is a system that manages a lot of data (messages, users, servers) that should stick around and be easy to search through!

Remembering to Not Forget: Databases

Spacebar uses a **relational database** to "write down" its in-memory objects. This gives rise to a couple of desirable properties:

- **Persistence** means the server can restart without losing data.
- **Queryability**, or the ability to efficiently answer questions about your data
 - "give me every channel where `guild_id = X`"
 - "find all servers these two users have in common"
 - "list all of the channels that a new user can see in this server"

Tables, Schemas, and Rows

A **relational database** organizes data into **tables**. Each table has:

- A fixed set of named, typed columns (the **schema**)
- Any number of rows (the actual data)

From Type to Table Schema

Simplified(!) TS Type

The fields of a type define what data a guild can hold.

```
interface Guild {  
  id: string;  
  name: string;  
  owner_id: string;  
  permissions: bigint;  
}
```

Database Table Schema

Those fields become the **schema**, which is like the header of the table.

ID	NAME	OWNER_ID	PERMISSIONS
1	test1	abcd	29348399
2	librarians	mr_books	23748237
3	2500 studyin'	sharry	42098430

From Values to Table Rows

A specific guild's data — its ID, name, owner, permissions — becomes one row in the `guilds` table.

`guilds`:

ID	NAME	OWNER_ID	PERMISSIONS
1234...	CIS 2500 Server	5678...	2147483647
9012...	Study Group	3456...	104857600
7890...	Gaming	5678...	104857600

ORMs: Define Schemas Using Types

An **ORM** (Object-Relational Mapping) translates between the database's "rows and columns" world and your code's "objects with fields and methods" world.

Spacebar uses **TypeORM** to do this.

The next few slides show off the syntax and some neat features. Don't worry about memorizing these! We just want to get comfortable with the basic idea for now.

TypeORM

A class decorated with `@Entity()` becomes a table. Fields decorated with `@Column()` become columns.

```
@Entity({name: "channels"})
class Channel {
    @Column()
    id: string;

    @Column({ nullable: true })
    name?: string;

    ...
}
```

channels

ID	NAME	...	...
1	Goofy Goobers	...	...
2	null	...	...
3	2500 Studyin'	...	...
8	announcements	7071	...

TypeORM

`@RelationId()` allows us to specify that one of the columns works like a link to another table.

```
@Entity({name: "channels"})
class Channel {
    ...
    @Column({ nullable: true })
    @RelationId(
        (channel: Channel) =>
            channel.guild
    )
    guild_id?: string;
    ...
}
```

channels

ID	NAME	GUILD_ID	...
1	Goofy Goobers	7070	...
2	null	null	...
3	2500 Studyin'	7070	...
8	announcements	4444	...

channels

ID	NAME	GUILD_ID	...
1	Goofy Goobers	7070	...
2	null	null	...
3	2500 Studyin'	7070	...
8	announcements	4444	...

guilds

ID	NAME	OWNER_ID	PERMISSIONS
1234	CIS 2500 Server	5678...	2147483647
7070	Study Group	3456...	104857600
4444	Gaming	5678...	104857600

TypeORM

`@JoinColumn()` allows us to specify that one of the columns works like a link to another table.

```
@Entity({name: "channels"})
class Channel {
    ...
    @JoinColumn({ name: "guild_id" })
    guild?: Guild;
    ...
}
```

channels

ID	NAME	GUILD_ID
1	Goofy Goobers	7070
2	null	null
3	2500 Studyin'	7070
8	announcements	4444

No column created for the guild itself! If you want the guild information for the given channel, the RDBMS uses `guild_id` to go look that up in the other table.

Foreign Keys: Saving Space

`@JoinColumn()` helps us create **foreign keys**, which are formal ways of tying rows of one relation together to specific rows of other relations.

- Helps avoid data repetition since we don't reproduce the full guild data with *every* channel in that guild

Foreign Keys: Letting the System Work for You

`@JoinColumn()` helps us create **foreign keys**, which are formal ways of tying rows of one relation together to specific rows of other relations.

- Helps maintain data consistency in other ways, like deleting of a guild. (Look at `@ManyToOne()` with `{onDelete: "CASCADE"}` to see how this is implemented.)

Before Deletion

guilds

id	name
4444	Gaming
7070	Study Group
1234	CIS 2500

channels

id	name	guild_id
1	announcements	4444
2	Goofy Goobers	7070
3	2500 Studyin'	7070

After

guilds

id	name
4444	Gaming
7070	Study Group
1234	CIS 2500

channels

id	name	guild
1	announcements	4444
2	Goofy Goobers	7070
3	2500 Studyin'	7070

Relations Are Not Always What They Look Like

In TypeScript, `channel.guild_id` is free to access, but `channel.guild` is not!

```
// This returns channels, but permission_overwrites will be undefined  
const channel = await Channel.findOne({ where: { guild_id: "7070" } });  
console.log(channel.guild_id)           // prints 7070  
console.log(channel.guild.member_count) // prints undefined
```

- `channel.guild_id` is a true attribute of the object: `guild_id` is stored as a column in `channels`
- `channel.guild` is a reference to another row from another table that's not loaded unless requested—no `guild` column exists!

Getting That Other Relation

You need to explicitly load the other relation in order for its data to be accessible as a part of the object returned:

```
// This loads the relation
const channel = await Channel.findOne({
  where: { guild_id: "7070" },
  relations: ["guild"]
});
console.log(channel.guild_id)           // prints 7070
console.log(channel.guild.member_count) // prints 16 (for example)
```


Finally, a CRUDdy Acronym

Over the lifecycle of a record in your database, it will be manipulated in one of four ways: **C**reate, **R**ead, **U**ppdate, or **D**eleate. (CRUD!)

In Spacebar and many other web applications, HTTP endpoints expose the functionality for doing each of these things to your data:

- **POST** → Create
- **GET** → Read
- **PATCH** or **PUT** → Update
- **DELETE** → Delete

Recap: Let the System, Not Discipline, Do the Work

- **Architecture is how you manage complexity once a codebase outgrows one person's working memory** — holistic properties like security and consistency have to be designed in, not remembered per-endpoint.
- **Decomposability and deep modules** let an agent (or a person) work correctly on one part of the system without holding all of it in context — a small interface, a large hidden implementation.
- **Documentation is part of the architecture now**: ADRs are LLM-readable records of decisions, and tools like `roborev` can enforce them automatically against every new commit.
- **Databases and ORMs give your in-memory objects persistence and queryability** — TypeORM maps classes to tables, and foreign keys (`@JoinColumn`) let related data live in one place instead of being copied everywhere.

Further Reading

- [A Philosophy of Software Design](#) — Ousterhout (book; deep modules concept)
- [Hexagonal Architecture](#) — Ports and adapters pattern
- [Microservices](#) — Martin Fowler
- [Deep Modules and Information Hiding](#) — Ian Bull

Questions?

`zives@engineering.upenn.edu` · `sharry@engineering.upenn.edu`