

Last time

SOP

XSS

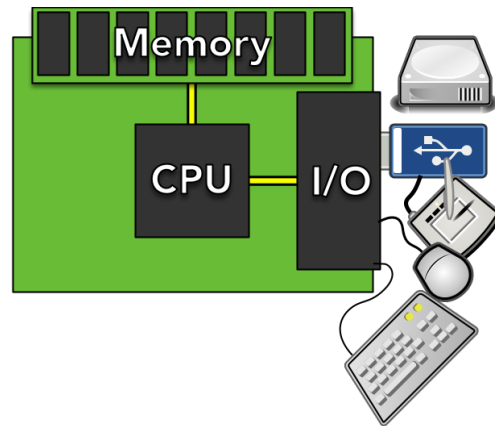
CSRF

Don't trust user input!

SQL injection

Another example of input validation failure

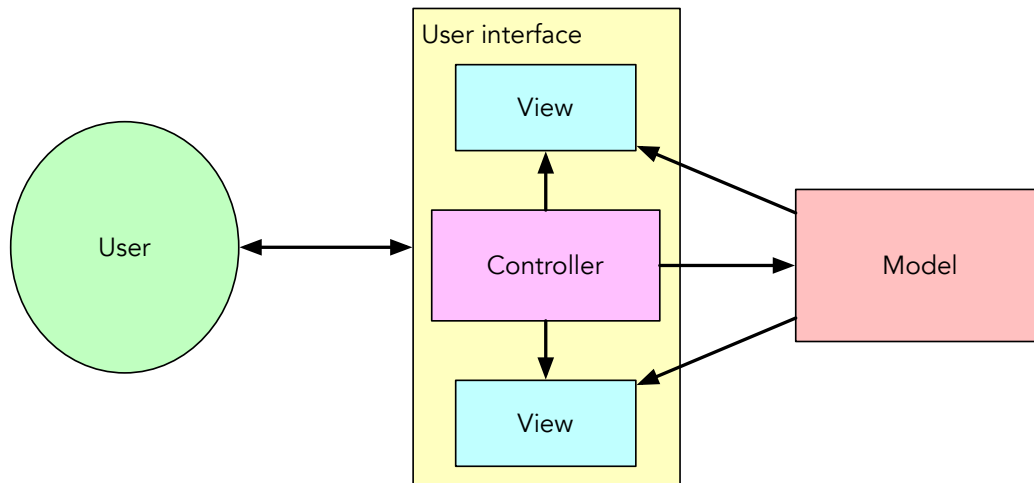
Why is this important?



4 / 27

Remember this simple (but _____) model of a computer? Let's take a look
simple model of a program.

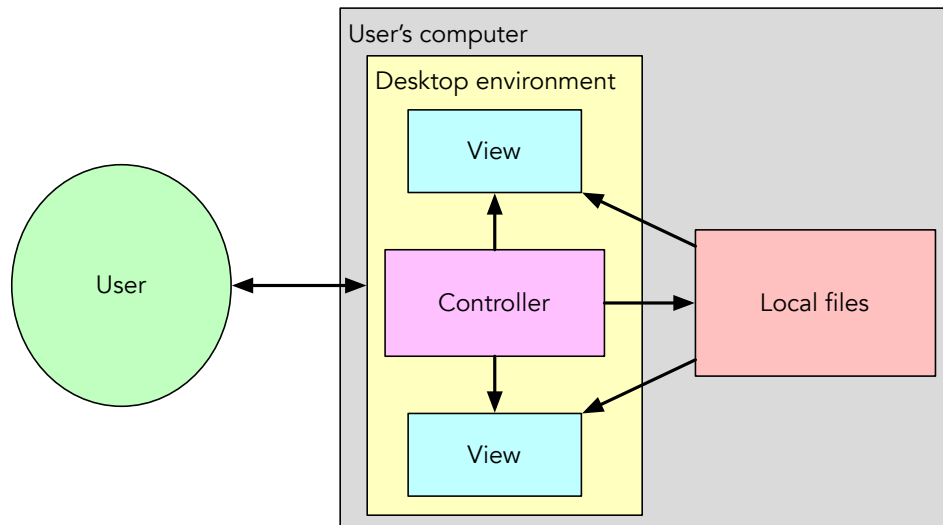
MVC pattern



5 / 27

You've probably seen the _____ (MVC) pattern in a Software Design course, as it's a useful way of structuring programs that users interact with. As a side note: you might not like the name, but it's better than the original name for Prof. Reenskaug's architecture: "thing-model-view-editor"!

Desktop MVC

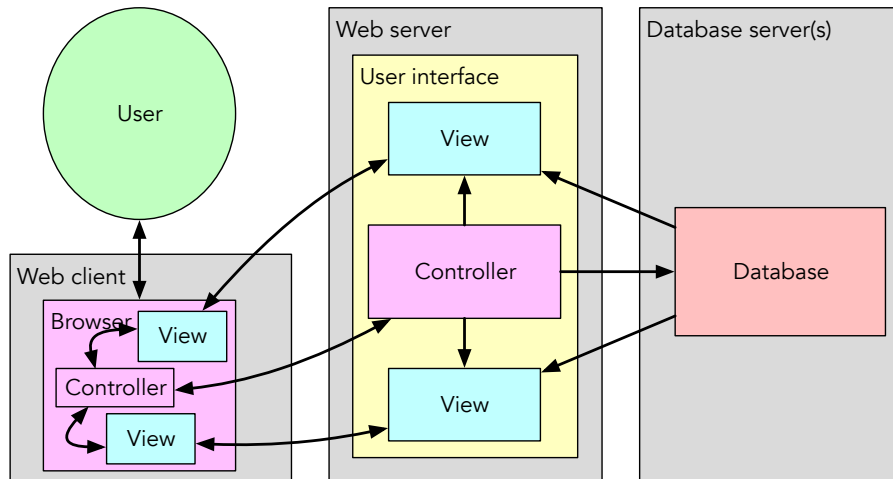


6 / 27

The MVC pattern was first designed for desktop computers, and it is very commonly used there. It also continues to appear in the cycle of technology trends in which everything centralized becomes distributed and everything distributed becomes centralized:

- remote access to applications via **X terminal**
- local ("native") applications (desktop, phones pre-iPhone)
- remote applications via the Web ("**there's no SDK!**")
- native applications on phones ("**what we call the App Store**")
- Web applications with native-like components (who needs native SDKs when you have React?)
- native applications based on Web technologies (React Native!)
- Web applications based on native application technologies that adapt Web application technology (React Native for Web)

Dynamic Web application



8 / 27

This is a more practical view of a real Web application. Within the browser, a rich application runs in JavaScript — and possibly WebAssembly — with code to control the UI and dynamic DOM-based view components that may talk independently to the backend on a web server. On that server, parts of the API may exist purely to serve view components, while other parts control what data is allowed to flow where.

At the back end of the backend, there is almost always some kind of persistent store: a database. It may be a traditional database with _____, or it may be an _____ database. It may be _____, it may have _____, but somewhere, data is stored. That data is often — not always, but often — accessed using the *lingua franca* of databases, the *Structured Query Language* (SQL).

Structured query language

Language for interacting with databases

ISO/IEC 9075

- SQL-92 (grammar)
- SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016, SQL:2019, SQL:2020+...

Not the only DB language (Cypher, GraphQL...), but...

9 / 27

SQL isn't the only language that can be used for accessing databases. Cypher is a query language used for accessing graph databases stored in the Neo4j database; it includes language syntax for dealing with nodes and edges and their relationship. GraphQL isn't, strictly speaking, a database query language any more than a REST API or an HTTP POST handler are. However, it is a language that a front-end can use to make queries about data stored in a back-end, so many of the same principles will apply!

SQL example

```
select * from student
  where
    discipline=(select id from discipline where name='ONAE')
    and student_id > 200800000
  order_by name
;
```

(probably better to do this with an INNER JOIN, but this isn't a DB course...)

Admin varies (MSSQL, MySQL, Oracle, PostgreSQL, SQLite...)

Core language should be consistent

10 / 27

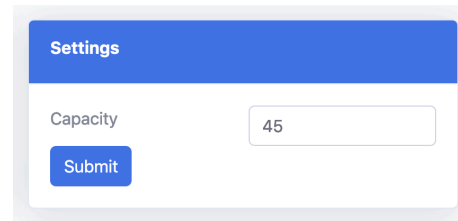
The way that you perform administrative actions like creating users or viewing a database's scheme can vary from database to database.

The fundamentals of how we perform the basic CRUD operations (Create, Read, Update, Delete) should be the same across all databases. Sometimes the performance of one approach vs another (e.g., a subquery vs an inner join) can be different on different platforms, but the semantics should be the same.

So what?

Example: Engineering match

- Need to track students, preferences, disciplines, capacities...



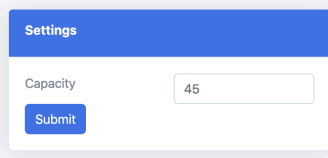
The image shows a web form titled "Settings". It contains a label "Capacity" next to a text input field that has the number "45" entered. Below the input field is a blue button labeled "Submit".

“*Hey, ECE is super-popular, let's add some more seats!*”

Updating ECE capacity

```
POST /admin/discipline/Computer HTTP/1.1
Host: localhost:5000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS
[...]
Content-Type: multipart/form-data; boundary=----1484928506128593855
Content-Length: 177
Cookie: mjx.menu=renderer%3ACommonHTML
-----1484928506128593855
Content-Disposition: form-data; name="capacity"

50
-----1484928506128593855
```



```
update discipline set capacity=50 where name='Computer';
```

Here's an example of how a UI element can drive an API request (in this case, an HTTP POST request) to cause an effect on the backend.

Q: what's missing from (the old version of) this request?

A: _____

At the backend, that request is parsed and turned into a SQL statement that, when executed, updates the database.

Web + SQL = all the things

Very common anti-pattern:

```
insert into readings(temp, humid) values (28.7, 83.9);
```

```
sql(f'insert into readings(temp, humid) values {temp}, {humid};')
```

```
update employees set name='Jonathan Anderson' where id=1234567;
```

```
sql(f"update employees set name='{new_name}' where id={employee.id};")
```

13 / 27

This is a very common pattern in Web development: we want to accept data from a user (or, more specifically, their user agent) and translate it into a SQL (or other) query to send to the backend database. However, it's actually an **anti-pattern**, for reasons that we will describe starting on the next slide...

The problem

```
f"update employees set name='{new_name}' where id={employee_id};"
```

```
POST /employee/1234567 HTTP/1.1
[...]
-----14849285061285938555493870477
Content-Disposition: form-data; name="name"

h4ckz0rd'; select * from employees; update employees set name='h4ckz0rd
-----14849285061285938555493870477--
```

```
update employees set name='h4ckz0rd';
select * from employees where name like '%';
update employees set name='h4ckz0rd' where id=1234567;
```

14 / 27

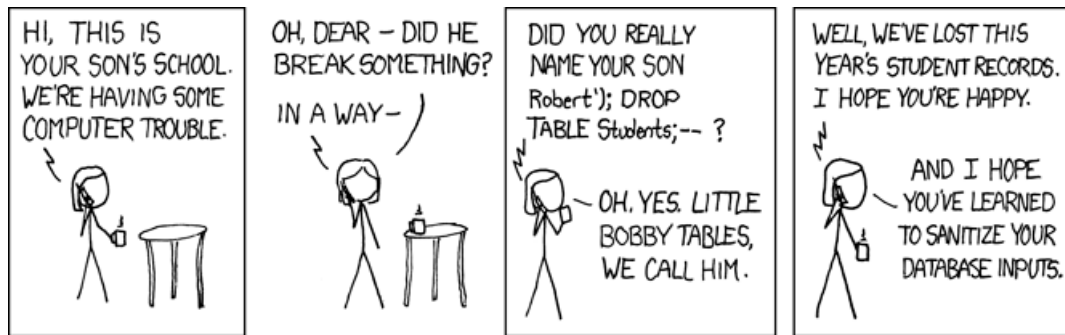
Let's say that we use the technique at the top of this slide to build a SQL query that updates an employee ID in a database.

Q: What's the problem with constructing SQL queries in this way?

Hint: in [the final part of Lab 5](#), you fuzzed a user's password, but you would also have observed something else: that there were some passwords which would cause the server to respond with neither an "access denied" message nor an "access granted" message, but something else entirely: a partial HTML page. You may have also noticed that some of these funny passwords contained a single-quote character ('). Why would that be?

Hint: What's the problem if we receive the **POST** request shown in the middle of this slide?

Untrusted input strikes again



Darn those humans... computing would be easier without them (???)

15 / 27

This is much like a program accepting input from a user (or a file) and passing it straight to a shell program.

What to do?

First described in **late 1990s**

No shortage of purported solutions*†^

* Boyd and Keromytis, "SQLrand: Preventing SQL Injection Attacks", in *ACNS 2004: Applied Cryptography and Network Security*, 2004. DOI: [10.1007/978-3-540-24852-1_21](https://doi.org/10.1007/978-3-540-24852-1_21)

† Halfond and Orso, "Combining static analysis and runtime monitoring to counter SQL-injection attacks", in *WODA '05 Proceedings of the third international workshop on Dynamic analysis*, 2005. DOI: [10.1145/1082983.1083250](https://doi.org/10.1145/1082983.1083250)

^ Buehrer, Weide and Sivilotti, "Using parse tree validation to prevent SQL injection attacks", in *SEM '05 Proceedings of the 5th international workshop on Software engineering and middleware*, 2005. DOI: [10.1145/1108473.1108496](https://doi.org/10.1145/1108473.1108496)

etc.

Common approaches

Scan for "bad" characters (e.g., ')?

- better hope you've thought of everything...
- ... and that you don't have any users from St. John's, NL

Parse with a SQL library?

Dynamic taint analysis?

21 / 27

There are several approaches that can be taken to try and detect "bad" inputs from users.

We've previously seen the limitations in scanning for "bad" characters. Besides, what if there is a legitimate need for a "bad" character, e.g., in St. John's? There are plenty of websites that won't allow townies to create accounts.

Some kind of string escaping *is* a good idea, but we want to be sure to apply it uniformly, with strong guarantees that it's done, not in an *ad hoc* "don't forget to escape the string over here too!" kind of way.

We know to be careful about using multiple parsers for the same language in security-critical code... subtle variations in parser semantics are **what hackers dream of!**

Dynamic taint analysis, while very cool-sounding, depends on things like your ability to produce sufficient test cases to dynamically check. Like fuzzing software, it may be a sensible risk mitigation technique, but it had better not be the only thing you're relying on!

Practical advice

If you find yourself writing:

```
query = query + f"where name = {user_supplied_value}"
```

Or even:

```
query = query + anything
```

Stop!

22 / 27

Essentially, application code should *never* manually construct commands that will be received as trusted input by something else. This is true for more than just SQL, but it's definitely true for SQL.

Note that we can't solve this problem by simply saying, "the database shouldn't trust SQL input". The application code running the database _____, so disallowing certain actions won't solve the fundamental problem. Instead, we need a different approach to building SQL statements.

ORM

Object-relational mapping

Tool for handling the tricky bits of SQL

You write object-oriented code

SQL experts write SQL query-building code

23 / 27

For SQL specifically, you should use an ORM instead of writing SQL statements. This allows for a better division of responsibility between the authors of application code and framework code.

ORM examples

Peewee (Python):

```
return (User
        .select()
        .join(Relationship, on=Relationship.to_user)
        .where(Relationship.from_user == self)
        .order_by(User.username))
```

diesel.rs (Rust):

```
update(users.filter(email.like("%@spammer.com")))
    .set(banned.eq(true))
    .execute(&connection)
```

Separation of concerns

Would you write your own TCP/IP stack?

Would you write your own crypto code?

Don't hand-craft your own SQL queries!

25 / 27

The answer to the questions on this slide should be, "maybe for fun, or on an assignment, but not in production!"

SQL injection

Web front-end / DB backend

SQL

SQL injection via untrusted input