Apps with users

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Section tomorrow: web app debugging
- Reading 4: due Friday, November 20, 11:59pm
- Project proposal: due Tuesday, November 24, 11:59pm
  - More info forthcoming!

Midterm debrief

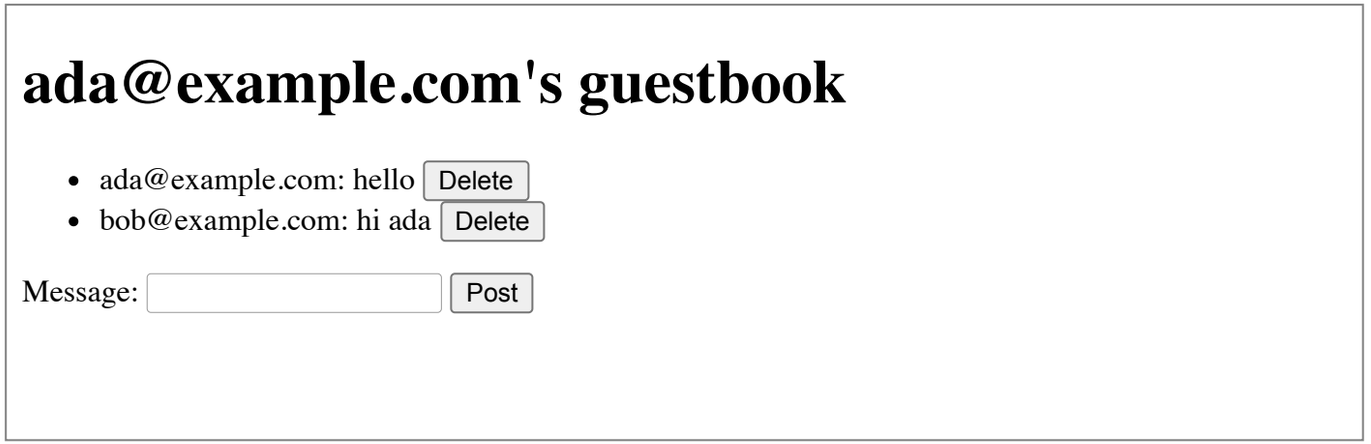
- A few exams are still being settled
- Median 77%, mean 76%
- Half the class scored between 69% and 85%
- Grades go up on Gradescope once the last exams are in
- If your final is higher, it replaces your midterm

One book for everyone



- Our guestbook with JavaScript in its page keeps one book
- Ada posts `hello`; bob opens the page and sees the same message
- Every page asks `GET /api/messages`, and gets the same answer
- The book is shared: anyone can leave a message, and anyone can see it

One book per user



1. Each user has a book at an address of their own: `/ada@example.com`
2. Any user who has signed in may read any book and sign it
    - The signature is their login, not a name they type
3. Only a book's owner may delete a message from it

Where does the program say one?

- List all places where the program assumes there is one guestbook
- **The field `book`, and its comment**: `Guestbook` keeps one `Book`
- **The path `/api/messages`**: nothing in the path says which book
- **The page's one list**: nothing on the page says whose book it is
- **The tests**: the post names no book, and the read expects the one

Half sheet, question 1 (3 minutes)

- Book is an ADT: a change to how its data is kept can stay inside it
- **Why is the change from one book to many not contained by the Book ADT?**

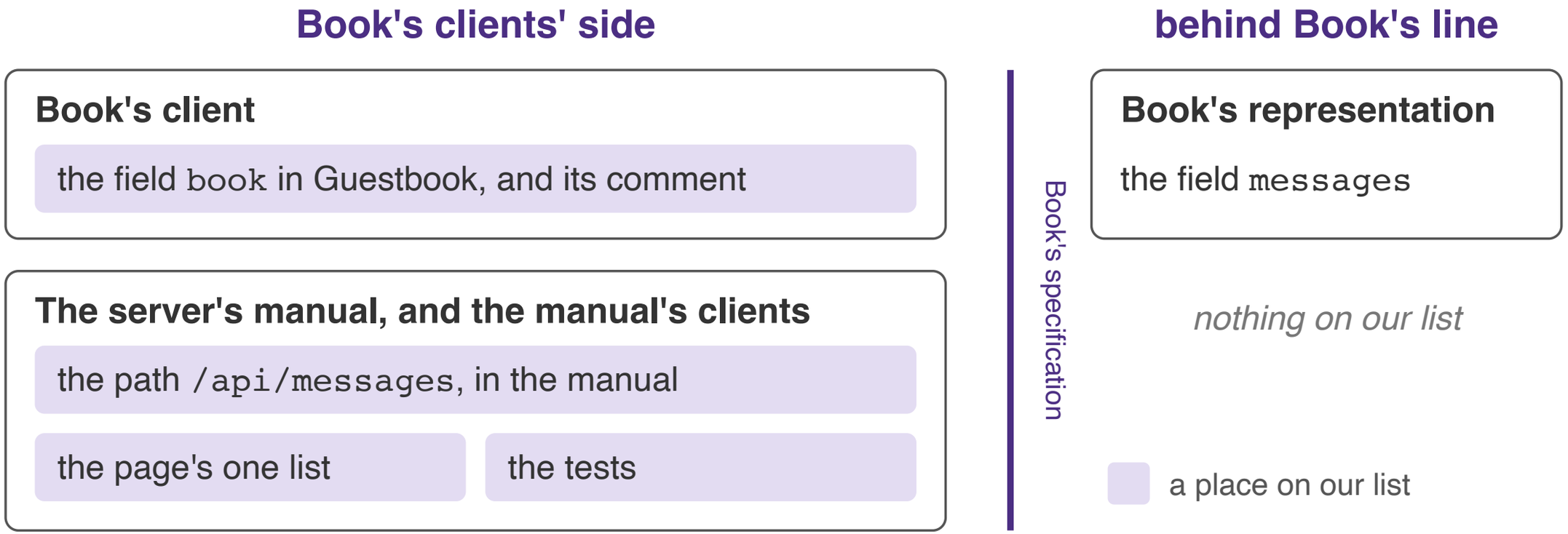
Why the change reaches everything

- That should be a surprise: an ADT can keep a change inside it

```
115   /** The messages posted so far, oldest first. */
116   class Book {
117       private List<String> messages = new ArrayList<String>();
```

- `Book`'s representation is its one field, `messages`
- That list says nothing about how many books there are
- Neither does the specification: we could just have more than one `Book`

The whole list is on the clients' side



- **Book's client:** `Guestbook` made one `Book`
- **The manual** promises "the messages posted so far": one sequence

A change to the abstract state

- The server's abstract state changes, to a book for each user
- Data abstraction keeps a client from depending on the representation
  - not on the abstract state: the client is meant to depend on that
- Reading 3: hide each decision likely to change inside a module
  - No module hides "one book": the manual promises it
- So the redesign writes the new abstract state first

A book for each user

- Something has to keep the books, and each book has to say whose it is
- So two ADTs where there was one: `Books`, new, and `Book`, changed

```
270   /**
271    * @return the book of the user whose login is owner, or an empty
272    *     Optional if that user has no book
273    */
274   public Optional<Book> bookOf(String owner) {
```

- `bookOf` looks up a book by its owner's login: there may be none

Book knows its owner

```
279  /** An owner, and the messages posted so far, oldest first. */
280  class Book {
```

```
297    public void add(User asker, String message) {
```

- A book is now an owner and a sequence of signed messages
- `User`: a user who has signed in, named by their login
- `add` signs the message with the login of `asker`, the user posting
- The earlier `add` signed with whatever name it was handed

The rest follows

```
25   *   GET /api/books/OWNER/messages
26   *                          OWNER's messages, as JSON
27   *   POST /api/books/OWNER/messages
28   *                          signs OWNER's book with the message the
29   *                          fields send, then OWNER's messages, as JSON
```

- The field holds one `Books`, where the earlier guestbook kept one `Book`
- The book is in the path now: `OWNER` stands for a login
- The page reads its owner from its own address, `/ada@example.com`

Testing: which book is now an input

- Bob signs ada's book `hi ada`, then opens his own:

bob, at `/ada@example.com`



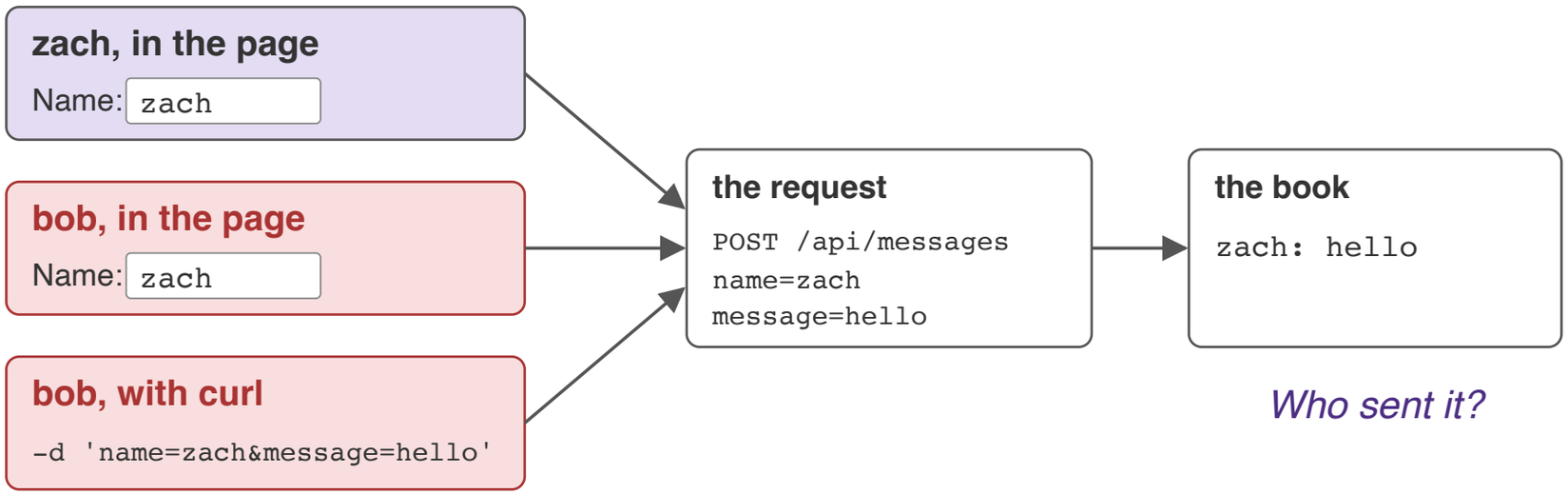
bob, at `/bob@example.com`



- Which book is now an input to every test: a test has to choose it

Which book, and who says?

- **The book**: any user may read and sign any book, so the client chooses
- **The user who sent it**: the earlier guestbook took it from the Name box



- Anyone can send those fields, but nobody may choose who they are

Authentication

Definition: authentication

Authentication is establishing who sent a request.

- "Establishing": evidence that the sender could not have made up
- The earlier guestbook was told "zach": it took a field's word for it
- "Who" is a user, named by a login such as `ada@example.com`
- **Testing**: who is asking is now an input, and one user is not enough

Who is asking?

- `handle` needs to know who sent every request, and asks `User.from`

```
223   public static Optional<User> from(HttpExchange exchange) {
```

- It returns the user who sent the request, or an empty `Optional`
- No method but `from` reads the header: procedural abstraction
- So how `from` finds out can change, from a header to a cookie, say

Checking passwords yourself

- The basic flow:
  - store a slow one-way function of each password, with a **salt**
  - sign in through a form, over HTTPS
  - start a **session**: a long random number, in a **cookie**
  - expire sessions, limit guesses, offer a way back in, ...
- This is hard to get right! So hand the job to someone else:
  - an **identity provider**, or a web framework's session code

Sidebar: Tailscale

- One neat way to be handed who is asking, with no password code
- Tailscale connects devices into a private network, a **tailnet**
  - on sign-in, "an identity provider authenticates the user"
- `tailscale serve 9331` gives the guestbook an address in the tailnet
- To each request, Serve adds a header, `Tailscale-User-Login`
  - filled with the requester's login name
- On the slides, `curl -H` sends that header itself, in Serve's place

Who may do what?

- One shared book had no owner: every visitor could do everything
- The third rule: only a book's owner may delete a message from it

Definition: authorization

Authorization is deciding what the user who sent a request may do.

- Authorization comes after authentication, and relies on it
- The rules are the program's own

Client-side authorization checks

```
36   const item = document.createElement("li");
37   item.textContent = messages[i] + " ";
38   if (me === owner) {
39     const button = document.createElement("button");
40     button.textContent = "Delete";
41     button.onclick = () => remove(i);
42     item.appendChild(button);
43   }
```

- `me`: the login the server sent back for `/api/me`
- `owner`: the login in the page's own address
- A Delete button only when `me` holds the login of the book's owner

In the browser, with both users

- Suppose the page decides alone. The server's `delete` takes no asker:

```
305   public void delete(int index) {
306       messages.remove(index);
307   }
```

ada posts two messages



bob opens ada's book, and posts



Half sheet, question 2 (3 minutes)

- The page draws a Delete button only when `me` holds the owner's login
- The server's `delete` takes no asker: it deletes for whoever calls it
- **The page shows bob no Delete button. How can bob still delete ada's message?**

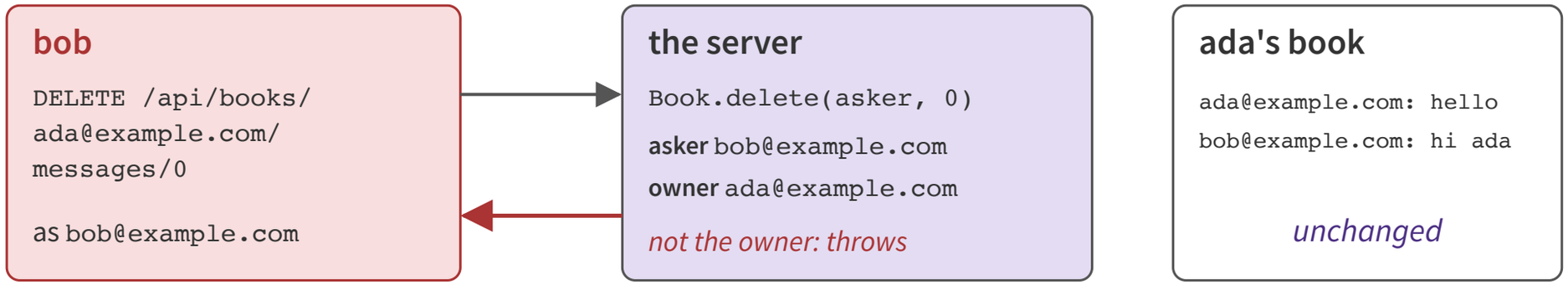
Bob, at the console, on ada's book

```
> me
"bob@example.com"
> me = "ada@example.com"
"ada@example.com"
> show()
undefined
```

- `show` draws a Delete button beside each of the three messages
- Bob clicks the first: status 200, and ada's `hello` is gone
- The check ran in bob's browser, on a variable bob can assign

The fix: decide on the server

- Started fresh: ada posts `hello`, and bob signs ada's book `hi ada`



403 `bob@example.com does not own this book`

- 403: the server understood the request, and refuses it

Where does the check live?

- In the handler, we would read every handler that reaches a `Book`
- Inside `Book.delete`, we read only `Book`:

```
308   public void delete(User asker, int index)
309           throws NotOwnerException {
310       if (!asker.equals(owner)) {
311           throw new NotOwnerException(
312               asker.login() + " does not own this book");
313       }
314       messages.remove(index);
315   }
```

Only Book to read

- With the comparison here, the code to read is Book. Two facts:
- **`delete` cannot be called without an asker**: a call has to pass one
- **No other method removes a message**:
  - `messages` is private: in Book, one line adds, one line removes
  - and `messages()` hands out a copy
- So every delete passes through the comparison
- A handler written next year cannot delete without going through it

Is it right?

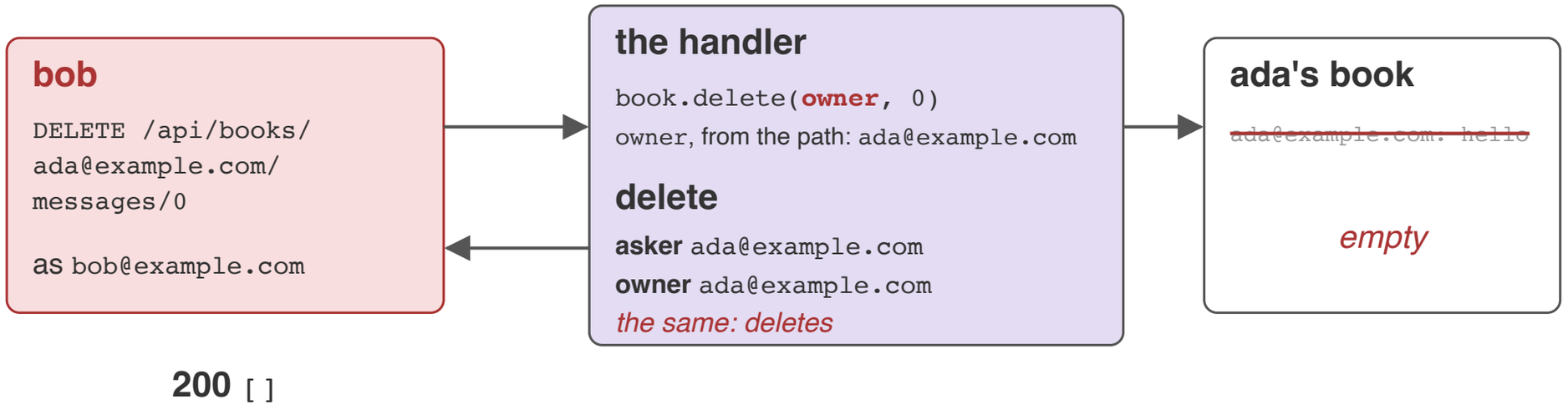
```
124   try {
125       book.delete(owner, index);
126   } catch (NotOwnerException e) {
```

```
308   public void delete(String asker, int index)
309           throws NotOwnerException {
310       if (!asker.equals(owner)) {
311           throw new NotOwnerException(
312               asker + " does not own this book");
313       }
314       messages.remove(index);
```

- **Here `asker` and `owner` are both a `String`. Is the call right?**

`javac` accepts it, and bob deletes

- `javac` accepts it, and prints nothing



- The handler passed `owner` where `asker` belongs
- So `delete` compared the owner with the owner: anyone deletes

User, a type of its own

```
209  class User {
210      private final String login;
211      // RI: login is not null, and not empty.
212      // AF(this) = the user whose login is login.
213
214      private User(String login) {
215          this.login = login;
216      }
```

- The constructor is `private`: only `from`, on a request, makes a `User`
- `add` and `delete` take a `User`. The owner in the path stays a `String`

The same mistake, in the finished program

```
125         book.delete(owner, index);
```

```
$ javac Guestbook.java
Guestbook.java:125: error: incompatible types:
    String cannot be converted to User
                book.delete(owner, index);
                            ^
⋮
1 error
```

- **We turned a coding mistake into a compile-time error.**
  - More on this on Wednesday, with design patterns!

The same change, in other forms

- A book for each user is today's case of "one becoming many"
- The same change comes in other forms:
  - a single-tenant app goes multi-tenant
  - a to-do app that kept one list lets each user keep several
  - an app for one device per user works on phone and laptop
  - an app for one currency, language, or time zone adds a second
- In each, a program written for one of something has to serve many

The ones that are left

- The finished program still assumes one of several things. Find some.
- **One owner for each book**: Book keeps its owner in one field
- **One signature on each message**: a message is one `String`
- **One book for each user**: open gives one only to a user who has none
- **One server**: a second copy, on a second machine, keeps its own books
- None of these is a bug: the rules asked for none of them to be many

One versus many

- **Notice the ones**: most were never decided, only assumed
- **Write them down**: a one a specification states, the next person finds
- **Know how to make one into many, later, if it is needed**:
  - find every place; sort by the ADT's line; write the abstract state first
  - let the rest follow; then ask what many brings
- **Reasoning**: the ADT's line said where the change had to go
- **Tools**: `User` turned a coding mistake into a compile-time error
- **Testing**: two books, two users, and requests the page never sends

Coming up

- **Tuesday**: section on web app debugging
- **Wednesday and Friday**: design patterns
- **Friday**: Reading 4 due, 11:59pm
- **Next Monday**: Reading 4 discussion, and upgrading apps
- **Next Tuesday**: project proposal due, 11:59pm
- Hand in your half sheet at the door