Reading 2 Discussion and Programs on the Web

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Tomorrow, Tuesday, October 20: section, on Floyd-Hoare logic
- Homework 2: due Friday, October 23, 11:59pm

Reading 2 discussion

- Form groups of 3 or 4 with the people next to you
  - Lamport readers: raise a hand, then spread out
- **Gabriel 1991**: ship half of the right thing, then improve it
- **Lamport 2003**: a program is a mathematical object; could you prove it?
- **Dijkstra 1978**: formality is a privilege; English makes nonsense easy

Share (5 minutes)

Knowing takes work: saying what a method must do, testing it, and reasoning about whether it does that. Is that work the job itself, or a tax on the job?

- Each of you, in turn:
  - Which article you read, and its answer
  - The question you wrote for the discussion

Gabriel's story, in this course's words

- Unix returns an error code, and every caller retries
- That is a method saying no by making its caller check
  - Lecture 6's question: what happens when a caller forgets?

Argue (6 minutes)

- **Gabriel**: ship half of the right thing; push the check onto the caller
- **The motto**: deliver code that you know works
- Who is right? What would your article say?
- If your group agrees, argue the other side, or take a case:
  - Binary search requires a sorted list: a fair contract, or a cop-out?
  - An insulin pump and a messaging app: the same philosophy, or not?
  - A chatbot that asks clarifying questions: is Dijkstra's objection gone?

Report (a few minutes)

- A couple of groups: what did you decide, and why?

Half sheet (3 minutes)

- The names of the others in your group
- What today's discussion changed in your thinking: job, or tax?
- One new point, and the article or person it came from
- Hand it in at the door as you leave

A guestbook

- A web page where a visitor leaves a name and a message
  - And every later visitor sees it
- 115 lines, on the program sheet; most of its Java is new to you
  - Start with the manual, lines 7 to 20
- The web of the 1990s: one page, one form, no JavaScript

Let's run it

```
$ javac Guestbook.java
$ java Guestbook
Serving at http://localhost:9331/
```

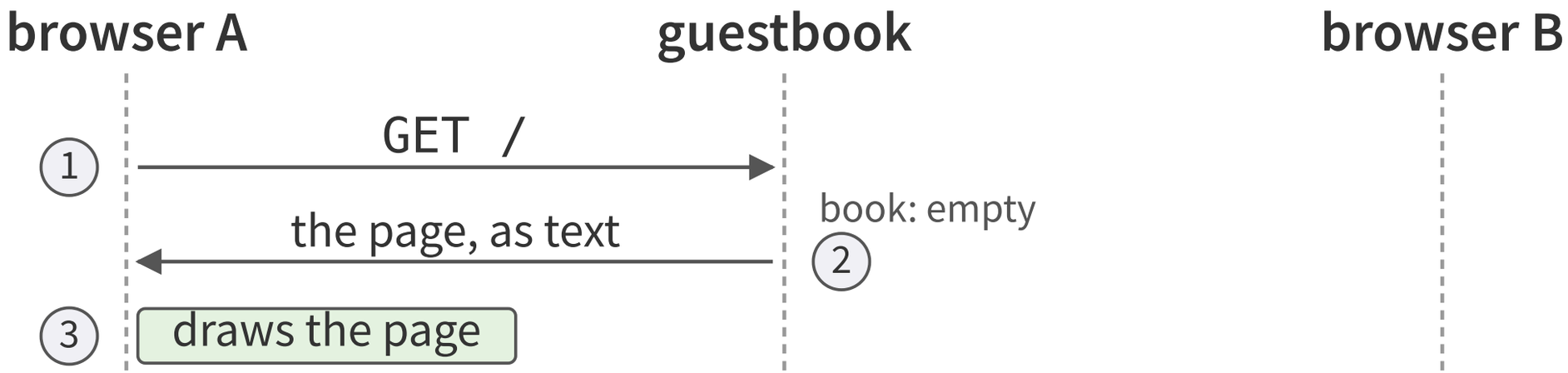
- Then nothing happens. What is the program doing?
- Waiting for someone to visit it

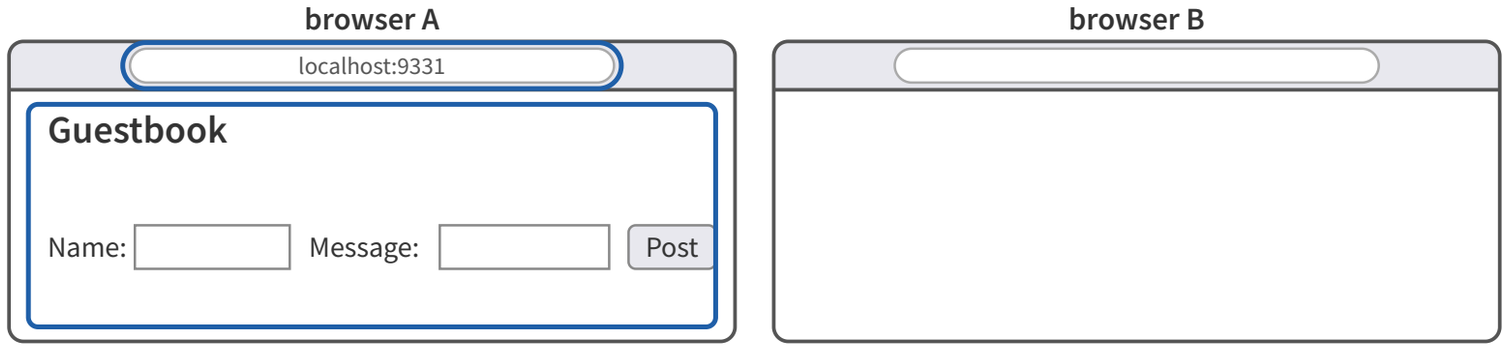
Visit it

- Open http://localhost:9331/: what will the terminal print?
- Post `Ada`, `hello world`: and now?
- A second window at the same address: what does it show, and why?

```
$ java Guestbook
Serving at http://localhost:9331/
GET /
GET /favicon.ico
POST /
GET /favicon.ico
GET /
GET /favicon.ico
```

What happened, step by step



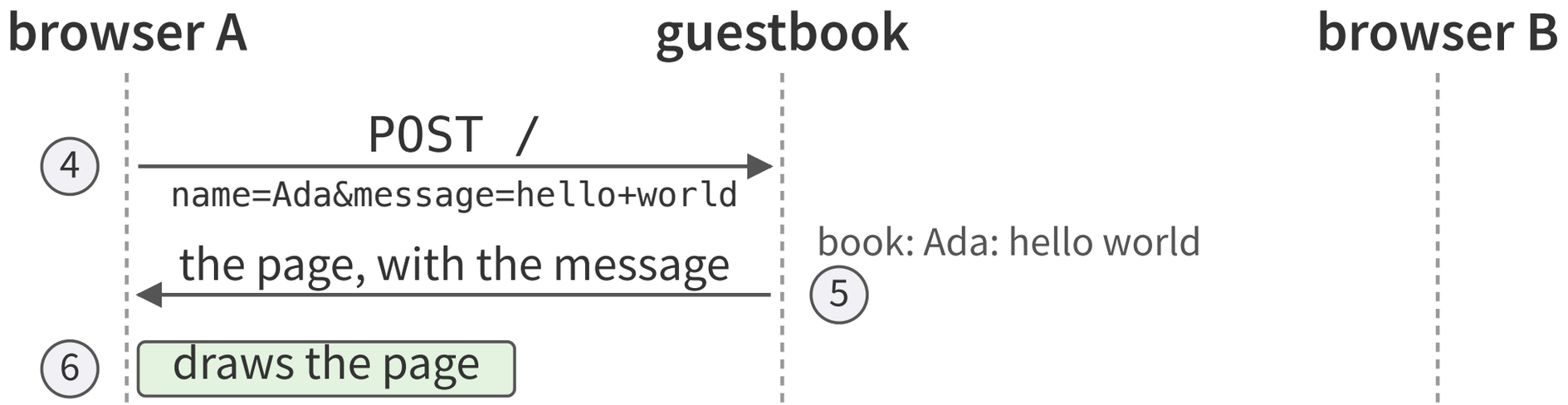


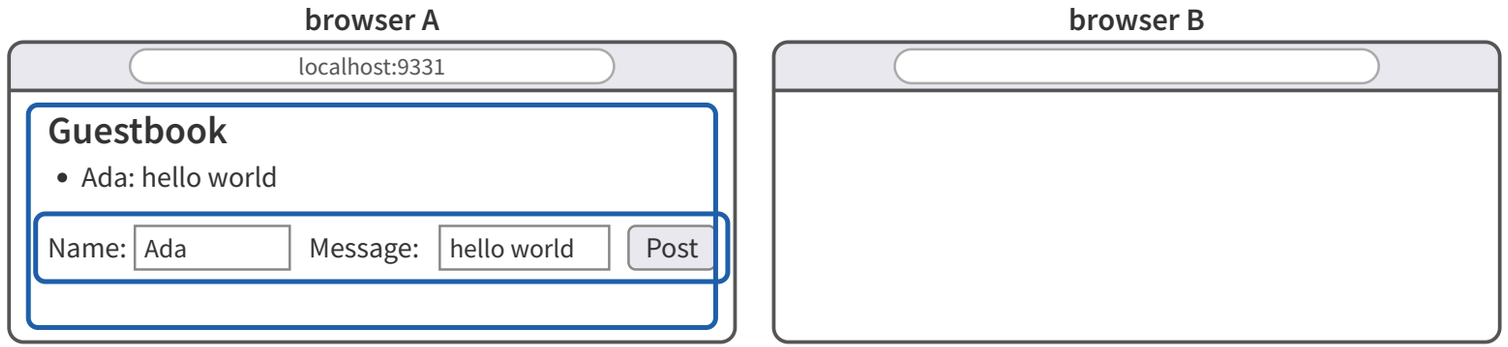
Browser A sends `GET /`, text; the terminal prints it

`handle` builds the page from the empty book, and sends it as text

Browser A draws the text: the heading, the boxes, the button

Posting a message



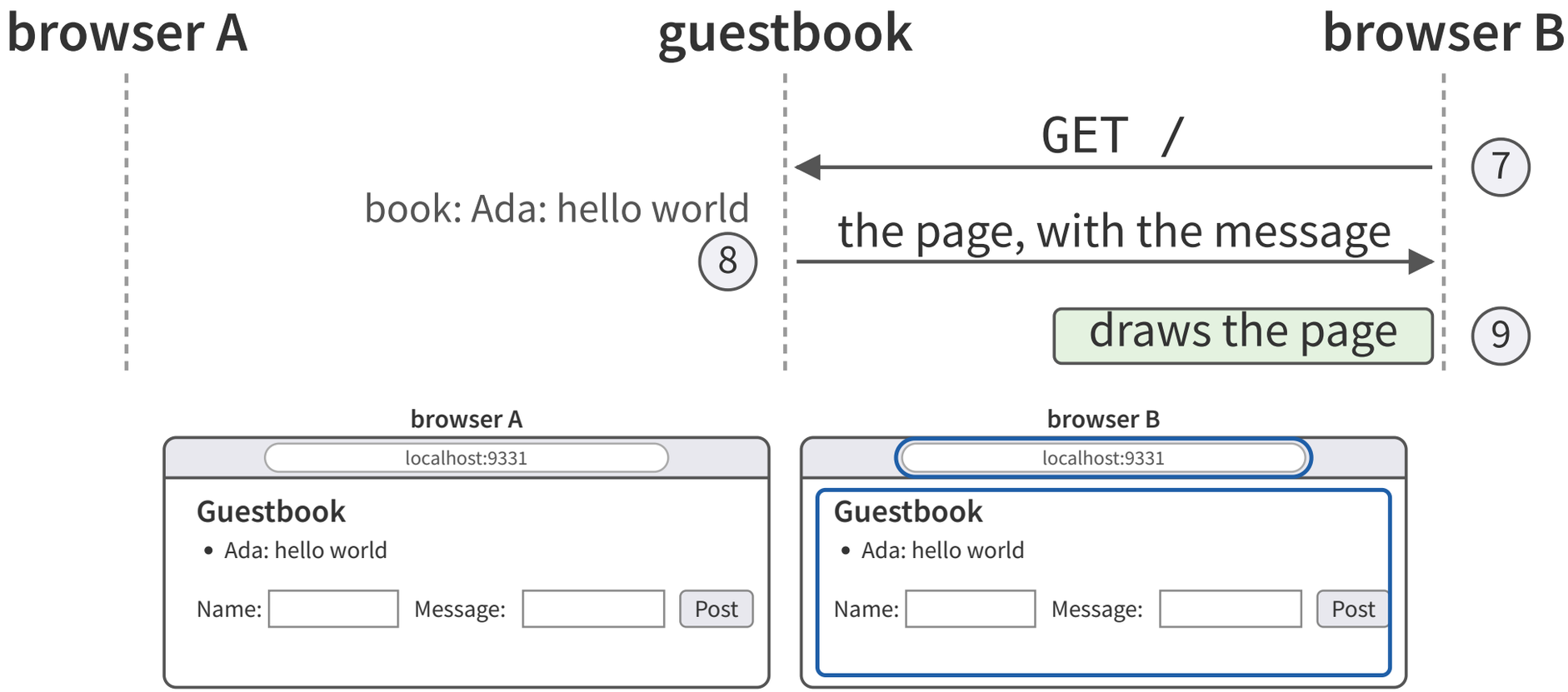


Post: `POST /`, ending with `name=Ada&message=hello+world`

`handle` adds the message, builds the page again, and sends it

Browser A draws the new page: the message, and empty boxes

The second window



Browser B sends its own `GET /`: the same line on the terminal

One book, so B's page is built with Ada's message in it

Browser B draws it: a message nobody typed in browser B

One book for everyone

Why did browser B see Ada's message?

```
21  public class Guestbook {
22      // The one book, shared by every request the server handles.
23      private static Book book = new Book();
```

- One `static` book: every request, from any browser, reads or adds to it
- **Lecture 6's three things**, each harder with other people
  - State that lasts; an order chosen by nobody; state you cannot see
- **A fourth**: the server does not know who it is talking to

What the browser sent, and who read it

```
GET / HTTP/1.1
Host: localhost:9331
```

```
41   String method = exchange.getRequestMethod();
42   System.out.println(method + " " + exchange.getRequestURI());
```

The method, `GET`, and the path, `/`: line 41 reads them, line 42 prints them

Which server the request is for; then lines about the browser, and a blank line

What came back, and who wrote it

```
HTTP/1.1 200 OK
Content-type: text/html; charset=utf-8
Content-length: 178

<html><body>
```

```
55  byte[] page = page().getBytes(StandardCharsets.UTF_8);
56  exchange.getResponseHeaders()
57      .set("Content-Type", "text/html; charset=utf-8");
58  exchange.sendResponseHeaders(200, page.length);
59  exchange.getResponseBody().write(page);
60  exchange.close();
```

A status, 200: the request was understood and answered

A header: what the body is, text in HTML

A header, the body's length: 178 bytes, counted after line 55 makes them

A blank line, then the body: the page, in HTML, text with tags

The POST, and who read it

```
POST / HTTP/1.1
Host: localhost:9331
Content-Length: 28
Content-Type: application/x-www-form-urlencoded

name=Ada&message=hello+world
```

```
46   if (method.equals("POST")) {
47       byte[] bytes = exchange.getRequestBody().readAllBytes();
48       String body = new String(bytes, StandardCharsets.UTF_8);
49       Map<String, String> fields = parseForm(body);
50       book.add(fields.get("name"), fields.get("message"));
```

The method is POST, so line 46 runs the branch

A body after the blank line, the form's fields: read as text

Split at & and =, decoded: + was a space

The message goes into the book; then the page, as for any request

Where is the loop?

```
25  public static void main(String[] args) throws IOException {
26      HttpServer server =
27          HttpServer.create(new InetSocketAddress(9331), 0);
28      server.createContext("/", Guestbook::handle);
29      server.start();
30      System.out.println("Serving at http://localhost:9331/");
31  }
```

The calculator's `main` was a loop. Seven lines here, and no loop: where is it?

Not a call: the method `handle` itself, handed to the server as a value

The loop starts inside `HttpServer`; `main` returns, and the program keeps running

Event-driven

- The loop is inside `HttpServer`: wait for an **event**, call `handle`
- **Event-driven**: methods called by a loop the program does not own
- **A library** is code you call; **a framework** is code that calls you
- `HttpServer` is a framework: nothing in the file calls `handle`, yet it runs
  - To know when, you read the framework's specification

Servers and browsers

A **server** is a program that waits for requests
and answers each with a response.

- The guestbook is a server; the browser is one of its users
  - The browser relies on what the guestbook promises
  - The guestbook can assume nothing about what the browser sends
- The web calls the browser the server's client
  - Lecture 2's client can be held to a promise; no browser can be

What can a user send?

- **Lecture 1's standard**: every input gets a described response
- The manual, lines 17 to 19, names two requests: `GET /` and `POST /`
- A browser sends other requests on its own
- `curl` sends whatever request it is told to: what else could arrive?

Your job is to deliver code that you know works.

Another path

```
$ curl http://localhost:9331/favicon.ico
<html><body>
<h1>Guestbook</h1>
⋮
```

Line 28: `handle` answers `/`, a prefix of every path: every path gets the page

- Harmless, but the manual does not describe it

A POST with no fields

```
$ curl -d '' http://localhost:9331/
⋮
<li>null: null</li>
⋮
```

`parseForm`'s map held neither field, so line 50 added `null: null`

- Lecture 6's seventh way: a made-up answer, on every visitor's page

A body that is not a form

```
$ curl -d 'name=%&message=x' http://localhost:9331/
curl: (52) Empty reply from server
```

`parseForm` requires a form's body, and nothing in `handle` checks: `URLDecoder` threw

- No response at all, not even the page

A message holding a tag

```
$ curl -d 'name=Eve&message=%3Cb%3Ehello%3C%2Fb%3E' http://localhost:9331/
⋮
<li>Eve: <b>hello</b></li>
⋮
```

Every later visitor's browser reads the tag: hello, in bold

- **Cross-site scripting**: `<script>` can run in every later visitor's browser

Refresh

What happens if I press refresh in browser A?

```
GET /
GET /favicon.ico
POST /
```

The browser asks first, then sends the POST again: the message twice

- A **GET** must not change the server's state; a **POST** may
- Browsers and crawlers send GETs on their own: the icon, every link
  - If a GET added a message, every crawler's visit would add a message

Coming up

- **Wednesday**: data abstraction
- **Friday**: data abstraction, continued
- Homework 2: due Friday, October 23, 11:59pm
- The notes' exercises close three of the four gaps the manual leaves
- Hand in your half sheet at the door