

SWE 621

SPRING 2021

ARCHITECTURAL STYLES

LOGISTICS

- ▶ HW2 due today
- ▶ Midterm in class next week during normal class period
- ▶ HW3 due in 3 weeks (3/23)
 - ▶ Will be released after midterm

MIDTERM

- ▶ 200 points / 20% of course grade
- ▶ ~80% based on lecture (including ideas covered in lecture and textbook)
- ▶ ~20% based on readings
- ▶ Open book / open Internet
 - ▶ However, all of your work must be your own, and all of your answers **MUST BE IN YOUR OWN WORDS!**
- ▶ 2 hours 40 minutes, during class time (4:30 - 7:10pm)
- ▶ Will be released at 4:30pm 3/9 via Piazza. Submit as Word doc or pdf through Blackboard.
- ▶ No need to use a drawing program to create any diagrams.

MIDTERM REVIEW

- ▶ Examples of questions
 - ▶ Questions on concepts, definitions, and process advice
 - ▶ e.g., What are the characteristics of a good abstraction?
 - ▶ Questions applying concepts to real world examples
 - ▶ e.g., critique this code snippet as an abstraction, based on this code scenario.
 - ▶ e.g., for these requirements, design a solution and describe through a component and connector model

IN CLASS EXERCISE

- ▶ Why might one build a software system organized into layers?

SOFTWARE ARCHITECTURE

- ▶ Software architecture = { Elements, Constraints, Consequences }
- ▶ Elements: the set of structures needed to reason about the system
- ▶ Constraints: the ways in which functionality is assigned to elements and elements can be composed
- ▶ Consequences: the resulting properties of systems which conform to the constraints

FREQUENT ARCHITECTURAL REQUIREMENTS

- ▶ Performance: how fast is the system
- ▶ Reliability: how likely is the system to be available
- ▶ Scalability: how well does adding more computing resources translate to better performance
- ▶ Maintainability: how hard is system to change
- ▶ Extensibility: in what ways can new components be added without changing existing components
- ▶ Configurability: how easily can the system behavior be changed by end-users
- ▶ Portability: in what environments can the system be used
- ▶ Testability: how easy is it to write tests of the system's behavior

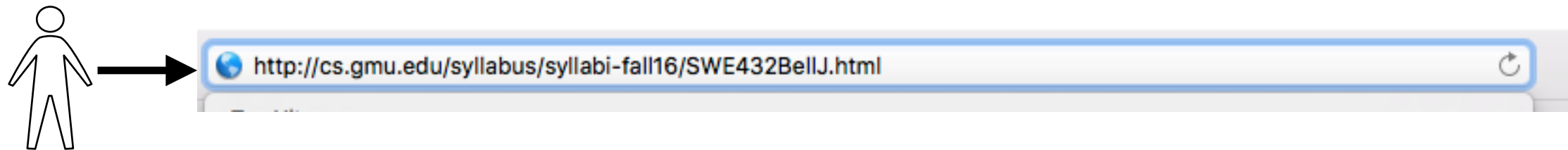
EXAMPLE OF ALTERNATIVE ARCHITECTURES: THE WEB

- ▶ Evolving competing architectures for organizing content and computation between browser (client) and web server
- ▶ 1990s: static web pages
- ▶ 1990s: server-side scripting (CGI, PHP, ASP, ColdFusion, JSP, ...)
- ▶ 2000s: single page apps (jQuery)
- ▶ 2010s: front-end frameworks (Angular, Aurelia, React, ...), microservices

STATIC WEB PAGES

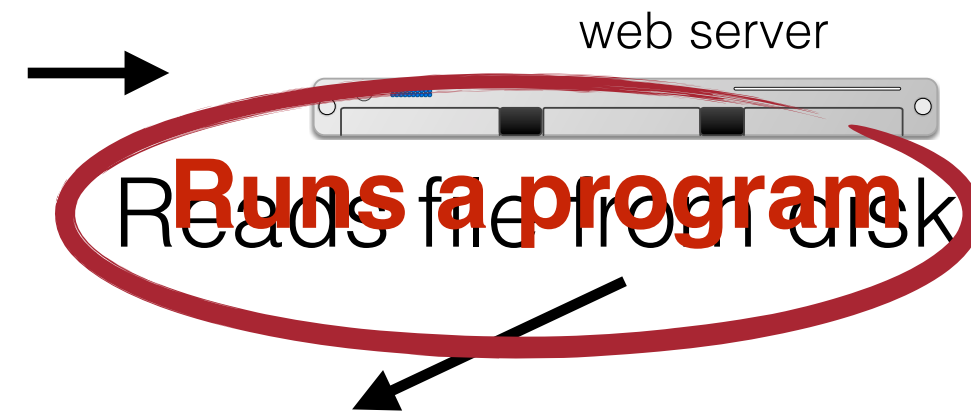
- ▶ URL corresponds to directory location on server
 - ▶ e.g. `http://domainName.com/img/image5.jpg` maps to `img/image5.jpg` file on server
- ▶ Server responds to HTTP request by returning requested files
- ▶ Advantages
 - ▶ Simple, easily cacheable, easily searchable
- ▶ Disadvantages
 - ▶ No interactivity

DYNAMIC WEB PAGES



HTTP Request

GET /syllabus/syllabi-fall16/SWE432BellJ.html **HTTP/1.1**
Host: cs.gmu.edu
Accept: text/html



HTTP Response

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head>...

SWE 432 Section 002 Fall 2016 Syllabus and Schedule

"Design and Implementation of Software for the Web"

Class Hours: Tuesdays and Thursdays, 12:00pm-1:15pm Robinson Hall B228

Grades, Readings available as pdfs: Blackboard

Resources (Announcements, Schedule, Assignments, Discussion):

Piazza - <https://piazza.com/gmu/fall2016/swe432001/home>

Instructor: Prof. Jonathan Bell

bellj@gmu.edu

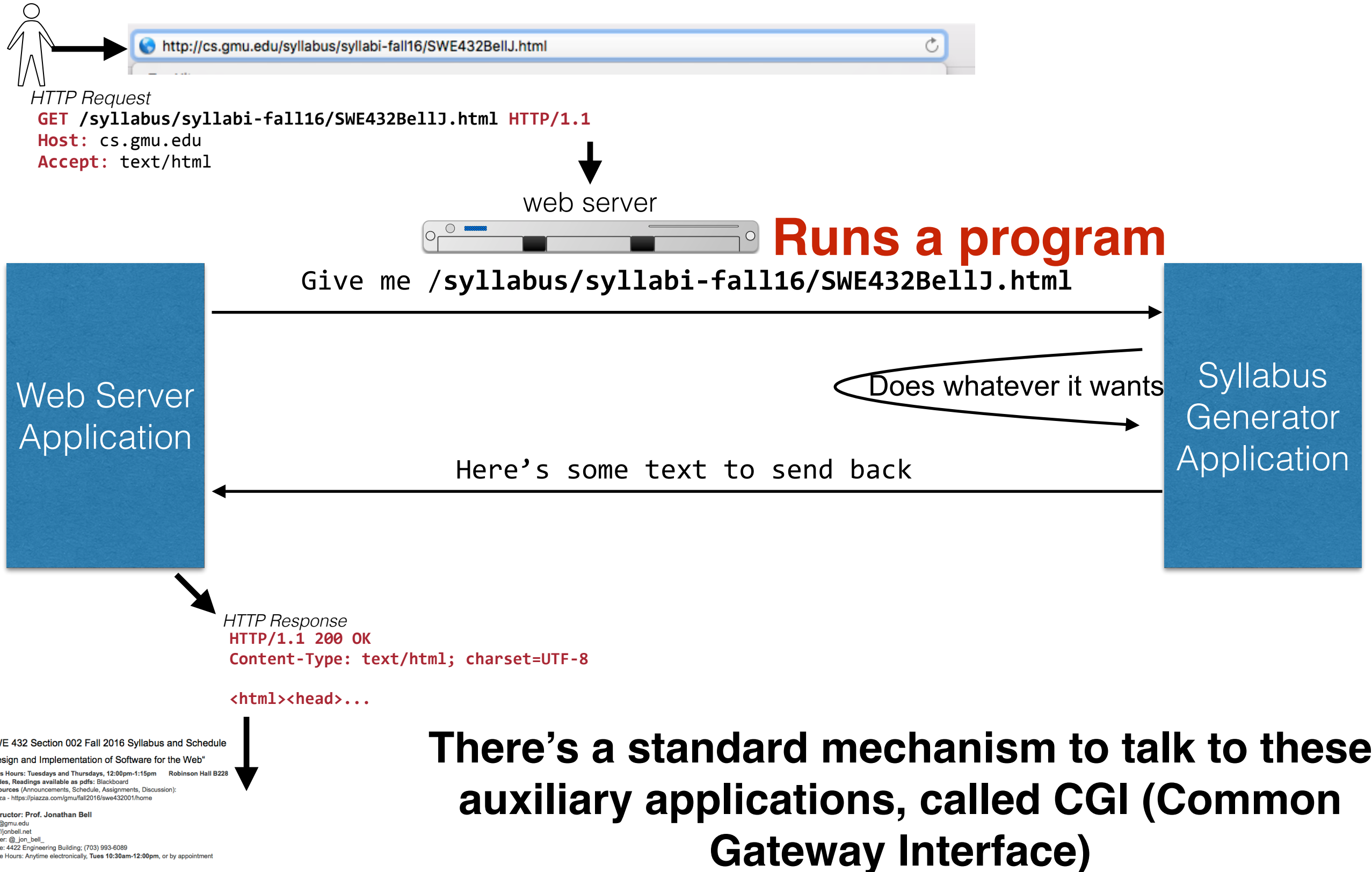
<http://jonbell.net>

Twitter: @_jon_bell_

Office: 4422 Engineering Building; (703) 993-6089

Office Hours: Anytime electronically, **Tues 10:30am-12:00pm**, or by appointment

DYNAMIC WEB PAGES



SERVER SIDE SCRIPTING

```
▶ ( <!DOCTYPE html>
  <html>
    <head>
      <title>PHP Test</title>
    </head>
    <body>
      <?php echo '<p>Hello World</p>'; ?>
    </body>
  </html>
```

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
  %>
    <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <%
    } else {
  %>
    <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
```

- ▶ Early approaches emphasized embedding server code *inside* html pages
- ▶ Examples: CGI

SERVER SIDE SCRIPTING SITE

Browser

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

HTML

*HTTP
Request*

*HTTP
Response
(HTML)*

Web Server

HTML templates, server logic, load / store state to database

Database



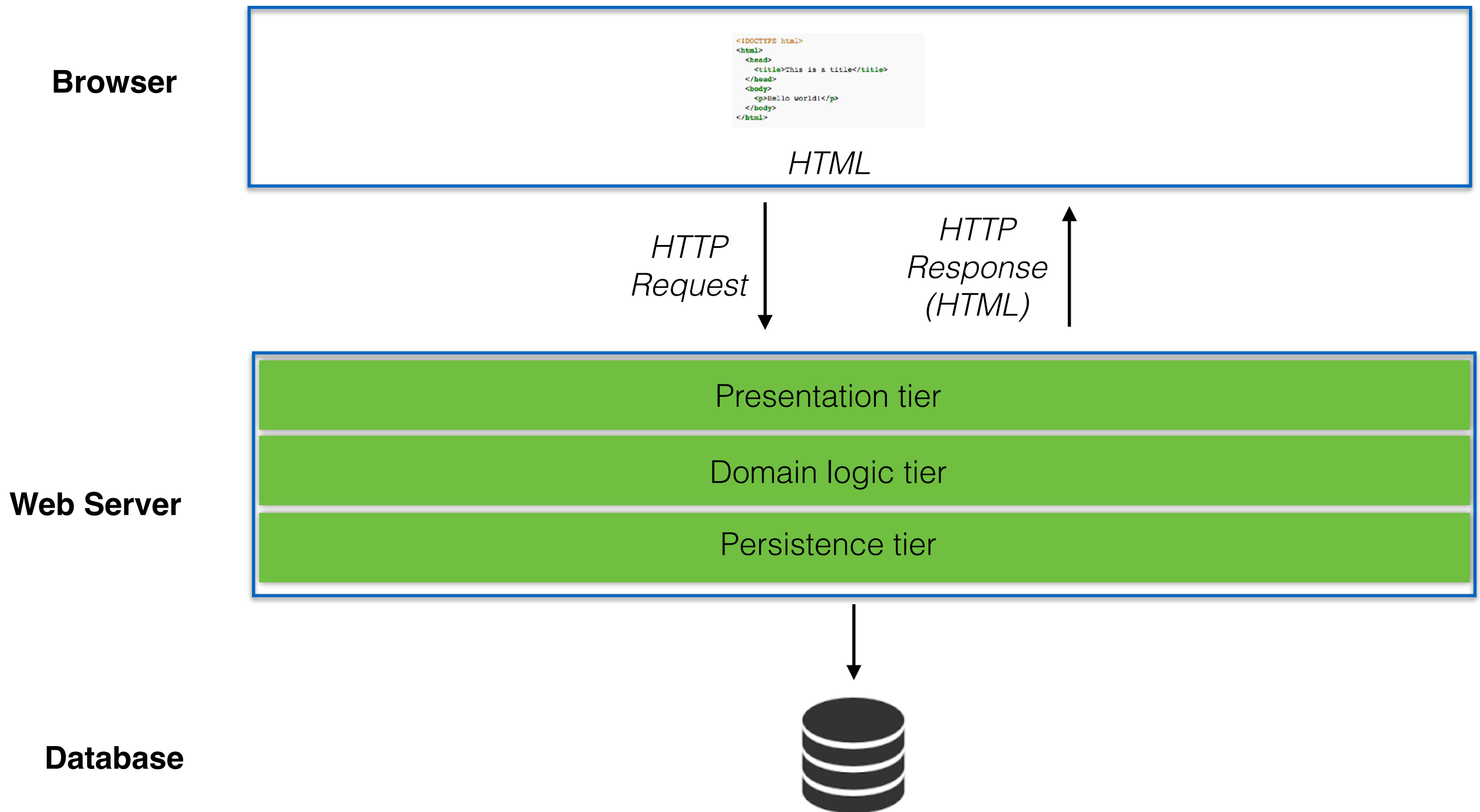
LIMITATIONS

- ▶ Poor **modularity**
 - ▶ Code representing logic, database interactions, generating HTML presentation all tangled
 - ▶ Hard to understand, difficult to maintain
- ▶ Still a step up over static pages!

SERVER SIDE FRAMEWORKS

- ▶ Framework that structures server into tiers, organizes logic into classes
- ▶ Create separate tiers for presentation, logic, persistence layer
- ▶ Can understand and reason about domain logic without looking at presentation (and vice versa)
- ▶ Examples: ASP.NET, JSP

SERVER SIDE FRAMEWORK SITE



LIMITATIONS

- ▶ Need to load a whole new web page to get new data
 - ▶ Users must *wait* while new web page loads, decreasing responsiveness & interactivity
 - ▶ If server is slow or temporarily non-responsive, ***whole user interface hangs!***
 - ▶ Page has a discernible *refresh*, where old content is replaced and new content appears rather than seamless transition

SINGLE PAGE APPLICATION (SPA)

- ▶ Client-side logic sends messages to server, receives response
- ▶ Logic is associated with a single HTML pages, written in Javascript
- ▶ HTML elements dynamically added and removed through DOM manipulation

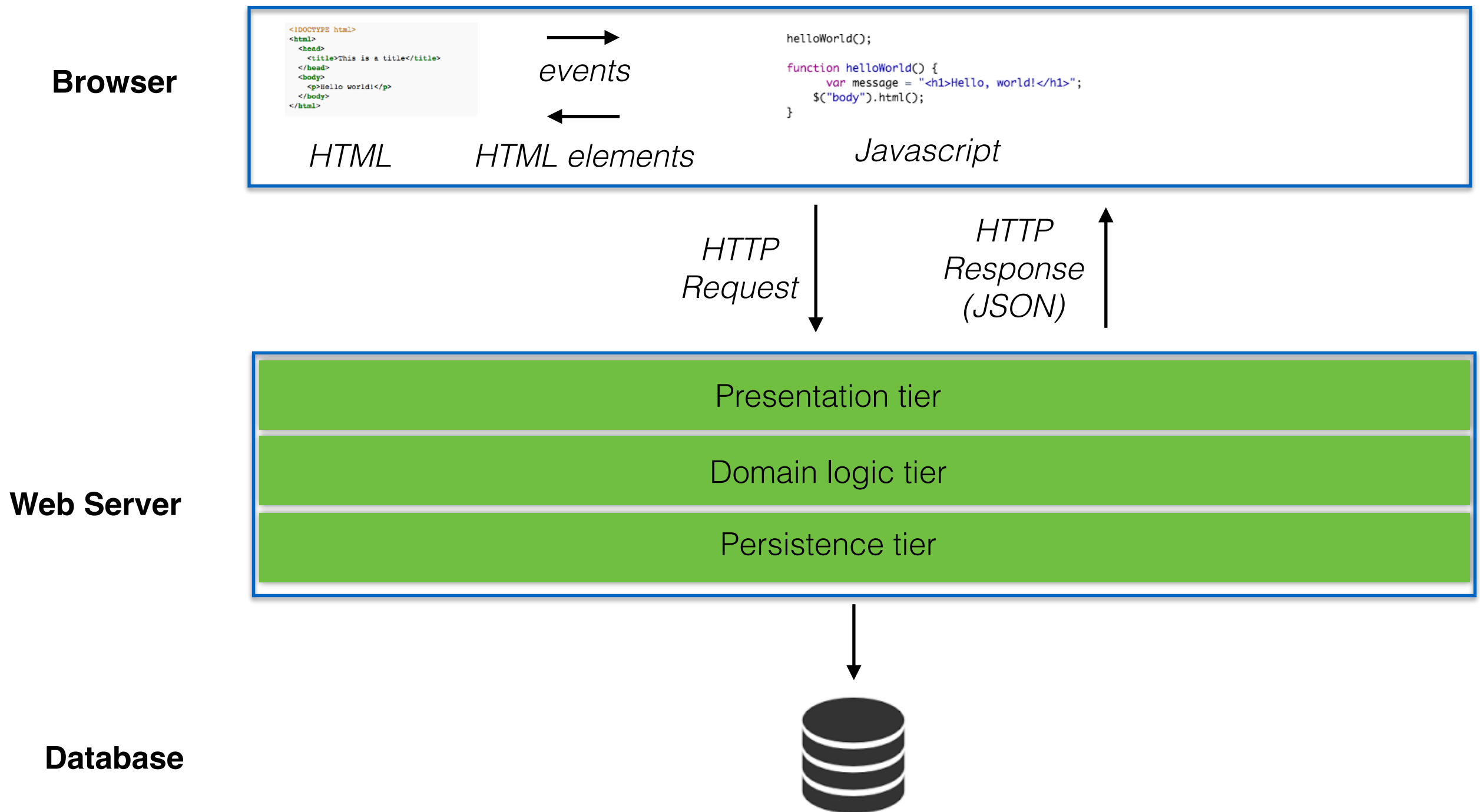
```
<b>Projects:</b>
<ol id="new-projects"></ol>

<script>
$( "#new-projects" ).load( "/resources/load.html #projects li" );
</script>

</body>
</html>
```

- ▶ Processing that does not require server may occur entirely client side, dramatically increasing responsiveness & reducing needed server resources
- ▶ Classic example: Gmail

SINGLE PAGE APPLICATION SITE



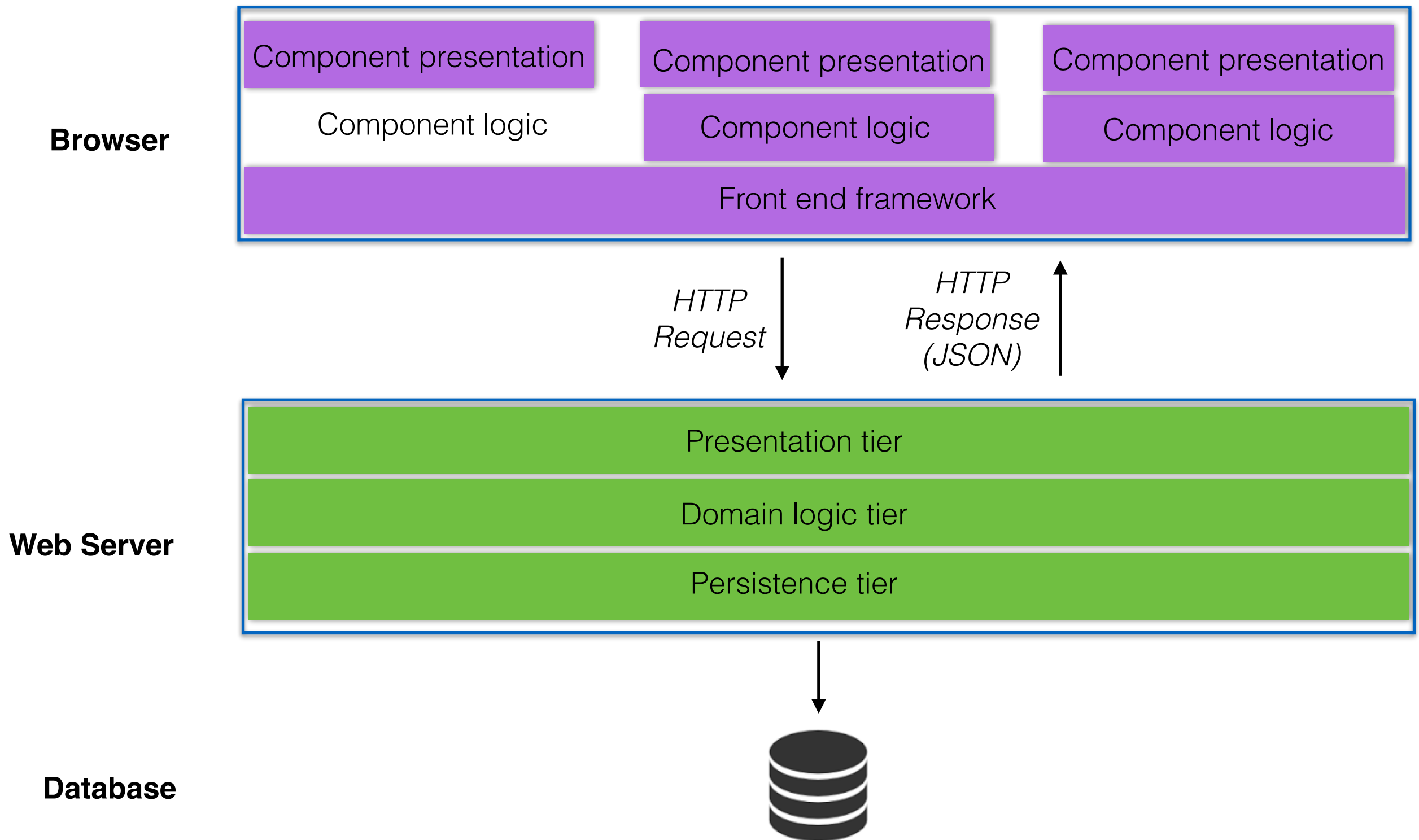
LIMITATIONS

- ▶ Poor modularity *client-side*
 - ▶ As logic in client grows increasingly large and complex, becomes Big Ball of Mud
 - ▶ Hard to understand & maintain
 - ▶ DOM manipulation is *brittle & tightly coupled*, where small changes in HTML may cause unintended changes (e.g., two HTML elements with the same id)
 - ▶ Poor reuse: logic tightly coupled to individual HTML elements, leading to code duplication of similar functionality in many places

FRONT END FRAMEWORKS

- ▶ Client is organized into separate *components*, capturing model of web application data
- ▶ Components are reusable, have encapsulation boundary (e.g., class)
- ▶ Components separate *logic* from *presentation*
- ▶ Components dynamically generate corresponding code based on component state
 - ▶ In contrast to HTML element manipulation, *framework* generates HTML, not user code, decreasing coupling
- ▶ Examples: Meteor, Ember, Angular, Aurelia, React

FRONT END FRAMEWORK SITE



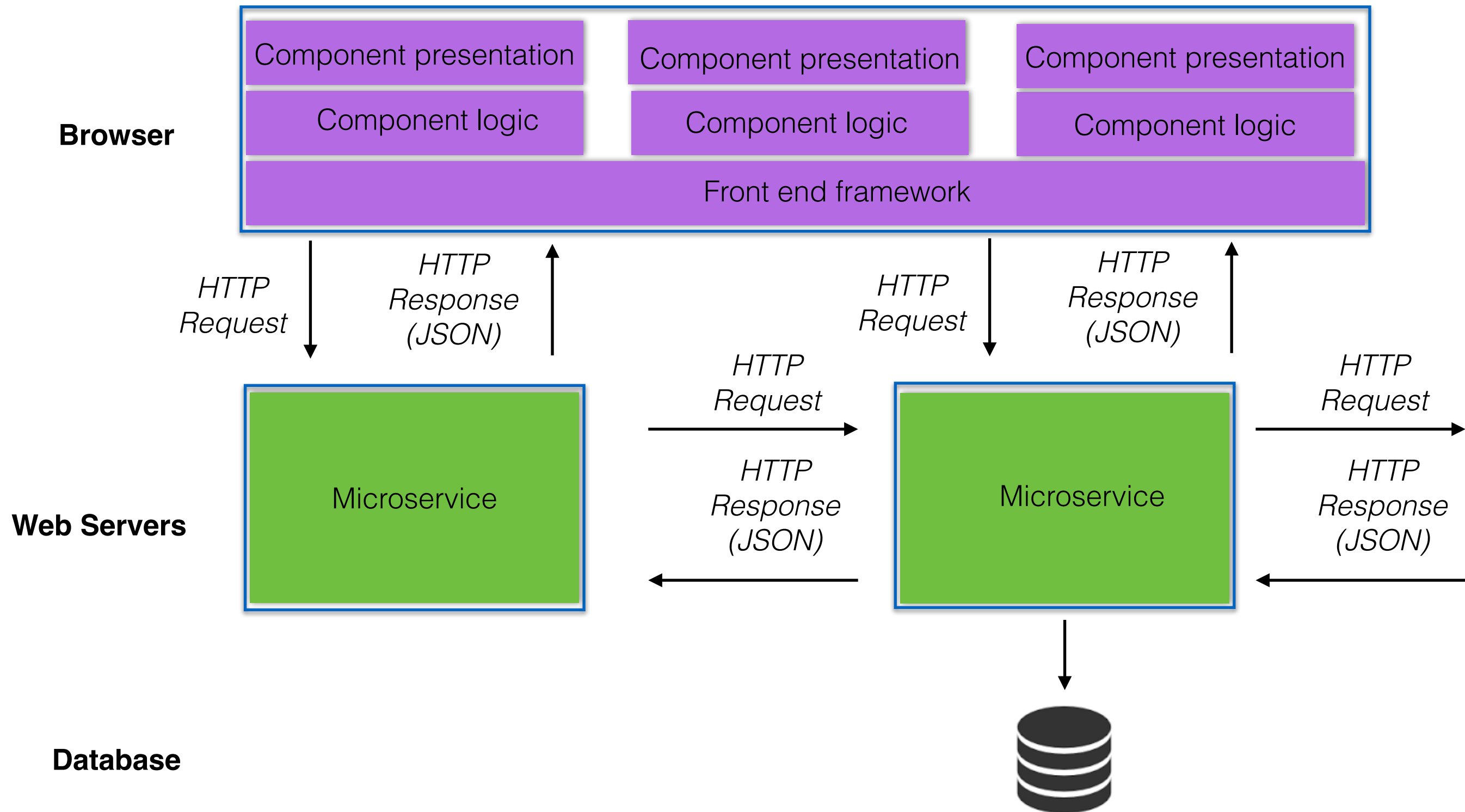
LIMITATIONS

- ▶ Duplication of logic in client & server
 - ▶ As clients grow increasingly complex, must have logic in both client & server
 - ▶ May even need to be written twice in different *languages!* (e.g., Javascript, Java)
 - ▶ Server logic closely coupled to corresponding client logic. Changes to server logic require corresponding client logic change.
 - ▶ Difficult to reuse server logic

MICROSERVICES

- ▶ Small, focused web server that communicates through *data* requests & responses
 - ▶ Focused *only* on logic, not presentation
- ▶ Organized around capabilities that can be reused in multiple context across multiple applications
- ▶ Rather than horizontally scale identical web servers, vertically scale server infrastructure into many, small focused servers

MICROSERVICE SITE



CAN WE DRAW MORE GENERAL LESSONS?

- ▶ Lots of different ways to organize a web app
 - ▶ Keep inventing new ones that are better by having some new properties
 - ▶ But may sometimes sacrifice others
- ▶ Can we draw any more general lessons about how to organize software?

ARCHITECTURAL STYLES

- ▶ Architectural style specifies
 - ▶ how to partition a system
 - ▶ how components identify and communicate with each other
 - ▶ how information is communicated
 - ▶ how elements of a system can evolve independently

ARCHITECTURAL STYLES

- ▶ Can also be characterized by one or more architectural decisions
 - ▶ e.g., elements in component A can send messages to elements in component B but not vice versa (i.e., layers)
- ▶ Making this decision(s) immediately has one or more consequences on architectural requirements
- ▶ Often binary
 - ▶ Either code conforms to the constraints and gains the consequences or has at least one violation and does not get the consequences

SOME COMMON ARCHITECTURAL STYLES

- ▶ Big ball of mud
- ▶ Layered
- ▶ Model-centered
- ▶ Publish/subscribe
- ▶ Pipe and filter
- ▶ REST
- ▶ Functional reactive programming

BIG BALL OF MUD

- ▶ Forces
 - ▶ Insufficient time to build the "right" way, with consideration of how design decisions impact maintainability
- ▶ Constraints: none
 - ▶ Anything can go anywhere.
 - ▶ Anything can be written in any way.
- ▶ Consequences
 - ▶ Leads to system that is disorganized.
 - ▶ Makes it hard to find where to make change, understand implications of change.
 - ▶ Decreases maintainability



<http://www.laputan.org/mud/>

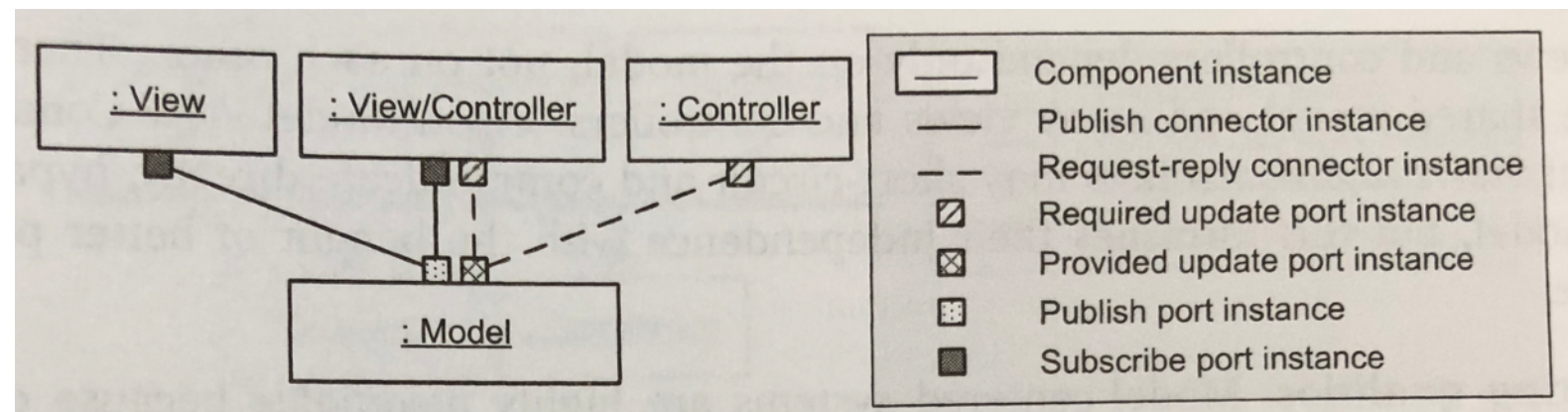
<http://www.laputan.org/mud/>

GMU SWE 621 Spring 2021

LAYERED ARCHITECTURE

- ▶ Elements: layers
- ▶ Constraints: can only use lower layers
 - ▶ Strictly layered: can only use adjacent lower layer
- ▶ Consequences
 - ▶ Supports maintainability by making it easier to find functionality
 - ▶ Supports portability and reusability by enabling layers to be swapped out

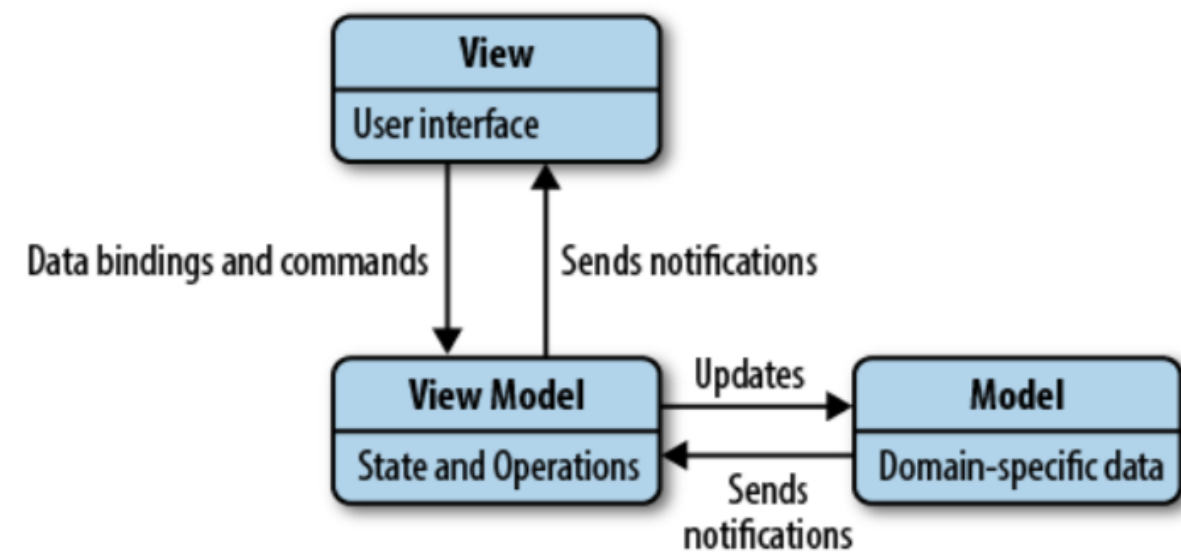
MODEL-CENTERED



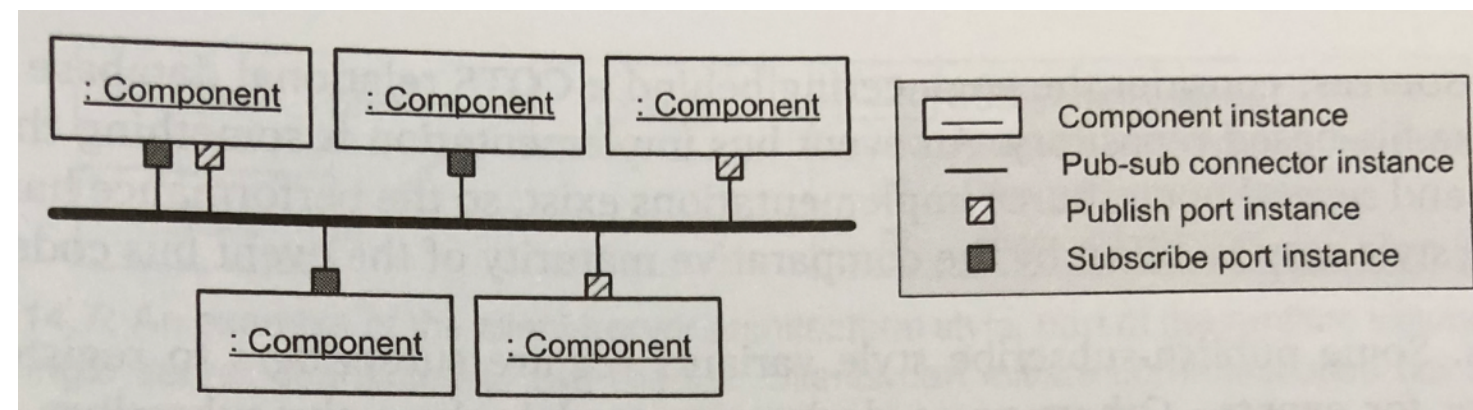
- ▶ Elements: model, view (optional), controller (optional), view-controller (optional)
- ▶ Constraints
 - ▶ Components interact with a central model rather than each other
 - ▶ Changes originates outside of model, propagate to model, trigger notifications to elements depending on model
- ▶ Synonyms: repository, shared-data, data-centered
- ▶ Consequences
 - ▶ Maintainable: can write data processing in terms of model rather than in terms of UI abstractions
 - ▶ Extensible: easy to add views, controllers, view/models without changing model
 - ▶ Scalability: can run each element in a separate thread

EXAMPLE: ANGULAR 1.0 -- MVVM

- ▶ Model: domain-specific data, doesn't matter how much it's interact with
- ▶ View
 - ▶ Visual representation of current state of model
 - ▶ View does not communicate with model directly Models are much more dumb: no formatting, etc
- ▶ ViewModel: processes user input, translates into format which work for model



PUBLISH/SUBSCRIBE

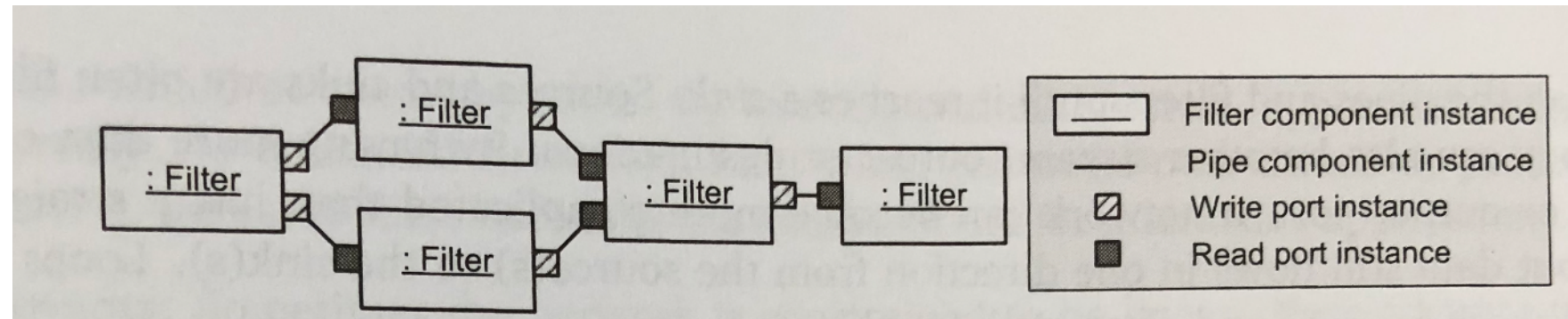


- ▶ Elements: component, event bus
 - ▶ Components broadcast events to listeners on event bus
- ▶ Constraints
 - ▶ Components do not know why an event is published
 - ▶ Subscribing components do not know **who** published event, depending on event type rather than specific publisher
- ▶ Synonyms: event-based, pub/sub
- ▶ Consequences
 - ▶ Maintainability: can make changes to components without impacting others
 - ▶ Performance: can (sometimes) reduce performance due to indirection

REST (REPRESENTATIONAL STATE TRANSFER)

- ▶ Elements: HTTP server, request / response connector
- ▶ Constraints:
 - ▶ Stateless: each client request contains all information necessary to service request
 - ▶ Cacheable: clients and intermediaries may cache responses.
 - ▶ Layered: client cannot determine if it is connected to end server or intermediary along the way
 - ▶ Uniform interface for resources: a single uniform interface (URIs) simplifies and decouples architecture
- ▶ Consequences
 - ▶ Scalability and reliability: enables servers to be added and removed at will at runtime
 - ▶ Performance: enables caching
 - ▶ Modifiability: hides changes behind URIs

PIPE AND FILTER



- ▶ Elements: pipes, filters, read ports, write ports
- ▶ Constraints
 - ▶ Filters may only interact through pipes
 - ▶ Filters may not share any global state
 - ▶ Filters may not make any assumptions about what happens upstream or downstream
 - ▶ Filter should incrementally read input and generate output
- ▶ Consequences:
 - ▶ Configurability, extensibility: can swap and compose networks of filters together, even at runtime
 - ▶ Scalability: can do computation in different filters in parallel
 - ▶ Modifiability: can more easily make independent changes

FUNCTIONAL REACTIVE PROGRAMMING

- ▶ Elements: component, stream of events
- ▶ Constraints:
 - ▶ Component only gets input from rest of system through stream of events; cannot access or mutate data elsewhere
 - ▶ When event arrives, changes state (resulting in new output) and may emit event to other components
- ▶ Consequences
 - ▶ Maintainability: much easier to make changes to individual element without having to think about consequences of that change to rest of system

SUMMARY

- ▶ Architectural style offer specific ways to achieve architectural requirements
- ▶ Often offer ways to separate functionality into separate elements and constraints on how these elements can interact
- ▶ Violating constraints of an architectural style often means that the consequences of the architectural style will no longer be realized

IN CLASS ACTIVITY

DESIGN ACTIVITY: TODO APPLICATION

- ▶ Form group of 3 or 4
- ▶ Your goal: design an architecture for a todo application by applying an architectural style (see next slide)
- ▶ Todo application requirements
 - ▶ User interactions with todos: add, delete, rename, complete, copy
 - ▶ Display todos to user
 - ▶ Persist todos
- ▶ Deliverables:
 - ▶ component and connector model showing elements in your system
 - ▶ explanation of architectural style, including discussion of constraints imposed on elements
 - ▶ text explaining how 3 different scenarios (e.g., add todo, copy todo, persist todos) are handled by the system, describing each scenario with details of what each component does

LIST OF ARCHITECTURAL STYLES

- ▶ Each group should select one of the following architectural styles.
- ▶ Architectural styles
 - ▶ Big ball of mud
 - ▶ Layered
 - ▶ Model-centered
 - ▶ Publish/subscribe
 - ▶ Pipes and filters
 - ▶ REST
 - ▶ Functional reactive programming

DESIGN ACTIVITY: STEP 2: DISCUSSION

- ▶ Compare and contrast designs based on each architectural style