Software Architecture

Spring 2019

(Some slides from Ben Liblit, UWisc CS 506)
Introduction

• **Software architecture** is the high-level structure and organization of a software system
  - The “big picture” or the “blueprint”: What are the components of the system, and how they fit together

• Useful reading on software architecture

• Let’s learn about one example architecture
  - Model-view-controller, a way of separating user interface from app’s domain logic

• Then we’ll talk more about architecture in general, and examine several more architectural styles
Model-View-Controller (MVC) Arch.

- **Model**
  - (state/database)
  - (business logic)
- **View**
  - (e.g., html)
- **Controller**
  - (business logic)
- **User**
  - (e.g., browser)

- Reads
- Reads/updates
- Renders
- Sent/shown
- Triggers action
Key Features: Model

• Model separate from user interface (view + controller)
  - Model need not concern itself with presentation
  - The core business rules can go into a model
    - Username cannot be null; email address must be well-formed; a credit to one bank account must correspond to a debit from another; etc
  - Ensures that the data always “makes sense”

• The app data is at rest in the model
  - One or more views will always see data consistently
  - One or more controller can access the model, transactionally
  - The model can be backed up!
Key Features: View

• View decides which data to show and how to show it
  ▪ But view should *not* change the model!

• Can have many different views
  ▪ Might vary based on user perspective, e.g., student sees their grade record one way, faculty sees it another way
  ▪ Might vary based on localization or customization, e.g., Fahrenheit vs. Celsius
  ▪ Might be shown simultaneously, e.g., individual slide view vs. thumbnail view in PowerPoint

• Views often linked to controller
  ▪ What the user can see informs them about what choices they have next, e.g., a student on a grade screen might be able to navigate to a courses screen
Key Features: Controller

- The input to the app
  - Provides actions that the user can invoke
  - Selects next view to display

- Business logic split between model and controller
  - E.g., checking whether current user has permission to read/write data could be done in controller or in model
  - Or, model could provide a method that includes the check, and then the controller calls that method
  - General design choice: include something in the model if multiple controller methods will reuse it

- Controllers often have a broad interface
  - E.g., in a web app, user could visit many URLs
  - E.g., in a desktop app, user could click many buttons
Example: Ruby on Rails

• Rails is a web app framework written in Ruby
  ▪ Developed by David Heinemeier Hansson as part of Basecamp; released separately as Rails in 2004

• MVC framework
  ▪ Model = database (sqlite, mysql, postgres, etc)
  ▪ View = `.html.erb` files, i.e., html with embedded Ruby
  ▪ Controller = methods that handle web requests

• Side note: real-world apps include many languages
  ▪ Ruby, HTML, CSS, JavaScript, SQL, …

• Very quick tour of Rails next, with some code
  ▪ Learn more at [https://guides.rubyonrails.org](https://guides.rubyonrails.org)
Sending a Web Request

• Browser sends a request for a web page

```bash
$ nc -c www.cs.tufts.edu 80
GET / HTTP/1.1
```

HTTP/1.1 200 OK
Date: Mon, 18 Feb 2019 20:57:47 GMT
Server: Apache/2.2.15 (Red Hat)
X-Powered-By: PHP/5.3.3
Content-Length: 848
Content-Type: text/html; charset=UTF-8
Connection: close

<html>
<head>
<title>Tufts University ECE and CS Departments</title>
</head>
</html>

(Type these three lines)
Rails Server Internal Sequence

- Server receives a request
- It first *routes* the request to a *controller* method
- That method accesses the db using *models*
- When the controller is done, it *renders* a view
- The view file is sent back to the web browser

- Note: HTTP is *stateless*
  - Each request connects, gets result, drops connection
  - Web server stores state in db and in browser cookies
  - Servers and OSes play a lot of tricks to avoid making so many connections
Rails Models

- Examples from *talks*, a web site for displaying a list of talks in the department

```
# db/schema.rb
create_table "talks" do |t|
  t.text "title"
  t.text "abstract"
  t.text "speaker"
  t.integer "owner_id"  # talk creator
end
```

```
# app/models/talk.rb
class Talk < ActiveRecord::Base
  validates_presence_of :owner  # an invariant!
end
```
Rails Routing: URLs to Methods

# config/routes.rb
Talks::Application.routes.draw do
  resources :talks
end

$ rake routes
  talks  GET  /talks(.:format)  talks#index
  POST   /talks(.:format)  talks#create
  new_talk  GET  /talks/new(.:format)  talks#new
  edit_talk  GET  /talks/:id/edit(.:format)  talks#edit
  talk  GET  /talks/:id(.:format)  talks#show
  PATCH  /talks/:id(.:format)  talks#update
  PUT    /talks/:id(.:format)  talks#update
  DELETE /talks/:id(.:format)  talks#destroy

• A route maps an HTTP verb and URL to a method
  ▪ Example: GET /talks/12 calls TalksController#show
Rails Controllers

• Receive a request

```ruby
# app/controllers/talks_controller.rb
class TalksController < ApplicationController
  def show
    # params maps :id to the id in the URL
    # Talk.find does a db query
    # @f for any f is a field (instance variable)
    @talk = Talk.find(params[:id])
    # notice the method just returns nothing!
  end
end
```
Rails Views

```erb
# app/views/talks/show.html.erb
<div class="center-header">
  <div class="talk">
    <div class="title">%= @talk.title %</div>
    <div class="speaker">%= @talk.speaker %</div>
  </div>
  <div class="abstract">
    <% if @talk.abstract == "" %>
      <span class="title">No abstract</span>
    <% else %>
      <span class="title">Abstract</span>
      <div class="abstract-body">%= @talk.abstract %</div>
    <% end %>
  </div>
</div>
```

- Didn’t need to def `Talk#title` and `Talk#speaker`
  - Implemented using reflection and knowledge of db!
MVC Pros and Cons

• Separation between data and interface is key
  - Views can be replaced, changed, customized, expanded
  - As db is read and written, changes reflected in all views
    - Centralized store of “truth” for the system state
  - Scalable deployment easier, e.g., multiple controller/view instances communicate with one db
    - System is quiescent when controller method returns

• Several potential drawbacks
  - Many kinds of changes to model require changing view and controller
    - E.g., removing a db column, changing the name of a table
  - Views and controllers are closely coupled
  - Added complexity
    - E.g., even simple Rails app has many files in many locations
What is Software Architecture?

- Tends to be more abstract than any coding feature
  - E.g., it may talk about things that are represented by sets of classes rather than a single class
- Is hard to change after building a system!
- Helps guide division of work by different developers
- Includes decisions, principles, and vision that led to the design
  - Informs later decisions as the system evolves
  - (Design patterns are smaller scale than architectures)
Pipe and Filter Architecture

• Each component has inputs and outputs
  ▪ Component reads input stream, produces output stream
  ▪ Components are filters, connections are the pipes
  ▪ A stream is a sequence of data of unknown length

• Key design properties/questions
  ▪ Filters should not share internal state
    - Filters don’t know how they’re connected
  ▪ At any joins, data rates from pipes need to sync up
  ▪ Filters and pipes have to agree on input/output data types
Pipe and Filter: Unix Commands

• Ex: Count `httpd` instances running (off by one)
  ```bash
  ps -ef | grep httpd | wc -l
  ```

• Commands take ASCII chars as input
  - What about unicode?

• Every command has standard in and standard out
  - But there’s also standard error, where does that go?
  - Normally to stdout, but you can redirect it

    ```bash
    # send both stdout and stderr to file.log
    ./script.sh > file.log 2>&1
    ```

• One command can launch another
  - See `pipe`, `fork`, and `exec`* C library functions
  - Most languages have a library to make these easier to use
Using Pipe and Filter in Java

- Standard input, output, and error are standard

```java
class System {
    static PrintStream out;  // stdout
    static InputStream in;   // stdin
    static PrintStream err;  // stderr
}
```

- To launch subprocesses, use `ProcessBuilder`

```java
ProcessBuilder pb = new ProcessBuilder("ls");
Process p = pb.start();
p.waitFor();
InputStream is = p.getInputStream();  // output of p!
BufferedReader br =
    new BufferedReader(new InputStreamReader(is));
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```
Pipe and Filter? A Compiler

• A compiler is a sequence of transformations
  - Source text converted to *tokens* and then parsed to yield AST
  - The AST becomes a control-flow graph (CFG), which is successively simplified
  - Ultimately the CFG is simplified so much it can be output as an assembly or machine code file

• Except, all stages share state (e.g., symbol table)
Pipe and Filter Advantages

• Overall behavior is a composition of filter behaviors
  ▪ Component connections are significant and obvious
• Potentially good reuse by creating different compositions of filters
• Can test each filter in isolation
• Filters can be replaced individually
• As we will see shortly, can support distributed execution
Pipe and Filter Disadvantages

- Not good for interactive use
  - Focused on converting inputs to outputs, not supporting user interactions
- Pipes are narrow; hard to pass complex data
  - E.g., compilers not really pipe-and-filter
- Overhead for parsing/unparsing data when read from/sent to a pipe
  - Though, components that are conceptually pipes could run as part of the same process
  - In which case they could pass data structures through “pipes”
Mapping a Stream

- A *map* operation transforms stream values

- One-to-one correspondence between elements
- Each element processed in isolation
- Input and output types may differ

\[
\begin{array}{c}
1 \rightarrow 3 \rightarrow 2 \rightarrow 9 \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\text{“one”} \rightarrow \text{“three”} \rightarrow \text{“two”} \rightarrow \text{“nine”}
\end{array}
\]

\[
(a \ stream)
\]

*map ints to Strings*
Reducing a Stream

- A \textit{reduce} operation combines stream values

- Many elements combined to yield a single value
- Elements processed with reduction of other elements
- Many possible reduction orders possible if operations are commutative and associative
- Input and output types may differ
- Stream must be finite to actually produce a value at the end
Google MapReduce

• Key challenge: Google data is BIG
  - Stored across vast number of machines in data centers
  - Need to process the data to do work
  - Can’t move all the data to one place; it won’t fit
  - Have to take machine failure into account
  - → programming a data processing computation is hard!

• MapReduce idea:
  - System architecture: map and reduce components only
  - Parallelize and distribute them across a data center
    - Sophisticated plumbing maps this fast, robust, fault tolerant
  • Pseudocode, pictures, examples shown next from Dean and Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004
MapReduce Pseudocode

• Count # word occurrences in documents

```
// map: (k1, v1) → List<(k2, v2)>
map(String key, String value) {
    // key: doc name, value: doc contents
    for (each word w in value) {
        EmitIntermediate(w, "1");
    }
}

// reduce: (k2, List<v2>) → List<v2>
reduce(String key, Iterator values) {
    // key: a word, values: list of counts
    int result = 0;
    for (each v in values) {
        result += ParseInt(v);
    }
    Emit(AsString(result));
}
```
MapReduce Types

• Slightly non-standard types for these operations
• Map input is key-value pairs, e.g., (doc, contents)
  ▪ (“US Constitution”, “We the People…”)
  ▪ (“Declaration of Independent”, “We hold these truths…”)
• Map output is a mapping written as a list
  ▪ [“We”, “2”), (“the”, “42”), (“People”, “1”), …] (numbers made up)
  ▪ [“We”, “5”), (“hold”, “3”), …] (numbers also made up)
• Reduce input is map output that is grouped by key
  ▪ (“We”, [“2”, “5”])
• Reduce output is result list for that key
  ▪ [“7”]
  ▪ It’s a list so it can be input to reduce further down the line
MapReduce Distribution

• Original data starts on many machines
• Those machines, or others, map that data
  ▪ Can be done massively in parallel, since mapping each piece of data is independent
• Reduce combines data across machines
  ▪ Can be done as a tree to maximize parallelism
Fault Tolerance

• If a worker times out, its work can be passed to another machine
  - Will only stall part of the computation
  - In many cases, worker’s result can just be ignored if it never finishes in time
  - After all, big data might have a lot redundancy and noise

• The Google File System stores reduce results
  - Large scale, robust, distributed file storage
Current State of MapReduce

- Open source tools available, e.g., Hadoop
- Programming model limited in many respects
- Google doesn’t use MapReduce any more
Layered Architecture

- Organized as a hierarchy
  - Layer provides services to layer above it
  - Layer is a client of the layer below it
- Example: running a Java program

<table>
<thead>
<tr>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system (kernel)</td>
</tr>
<tr>
<td>Java virtual machine</td>
</tr>
<tr>
<td>Java program</td>
</tr>
</tbody>
</table>
Open Systems Interconnection (OSI)

- “Network stack” or “protocol stack”

<table>
<thead>
<tr>
<th>Layer 7: Application (HTTP)</th>
<th>(the actual application)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 6: Presentation</td>
<td>(encoding, compression, crypto)</td>
</tr>
<tr>
<td>Layer 5: Session</td>
<td>(sequences of communication)</td>
</tr>
<tr>
<td>Layer 4: Transport (TCP)</td>
<td>(segmentation, acks, multiplex)</td>
</tr>
<tr>
<td>Layer 3: Network (IP)</td>
<td>(addressing, routing, etc)</td>
</tr>
<tr>
<td>Layer 2: Data link (Ethernet)</td>
<td>(sending data frames between nodes)</td>
</tr>
<tr>
<td>Layer 1: Physical (IEEE 802.3u)</td>
<td>(sending raw data over wires/radio/etc)</td>
</tr>
</tbody>
</table>
## LAMP Stack

<table>
<thead>
<tr>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
</tr>
<tr>
<td>Apache</td>
</tr>
<tr>
<td>Linux</td>
</tr>
</tbody>
</table>

- Slightly old-school web server structure
Layered Architecture Tradeoffs

• Advantages
  ▪ Good fit for system with increasing levels of abstraction
  ▪ Changing one layer affects at most two others
  ▪ Can interchange implementations of same layer interface

• Disadvantages
  ▪ May not be able to identify clean layers
  ▪ Might need to jump layers for performance or functionality
Client-Server Architecture

- Clients communicate with server, typically over network
- Tradeoffs
  - Server is central point of failure
    - Replication can help, but then consistency is
    - CAP theorem: Pick two of Consistency, availability, and partition tolerance
  - Any client-to-client communication must go through server
Peer-to-Peer Architecture

- Machines communicate with each other
  - Popularized by Napster (!), 1999
- Several challenges/tradeoffs
  - Trust between nodes
  - More equal upload/download volume compared to client server
  - Location of data on network not centralized
Software Architecture Activities

• Initial design/development/evaluation
  - Note architecture can have several different views
    - User perspective; development view (software modules); run-time view (processes etc); physical view (hardware, location in data center, etc)

• Maintain architecture over time
  - Architectural drift — implementation decisions that aren’t encompassed by the architecture, but don’t conflict with it
  - Architectural erosion — implementation decisions that actually violate the architecture
  - What to do?
    - Change the architecture or change the code!

• Evolve architecture as requirements change

• Key challenge: Architecture is not code, hence it will inevitably drift from the code