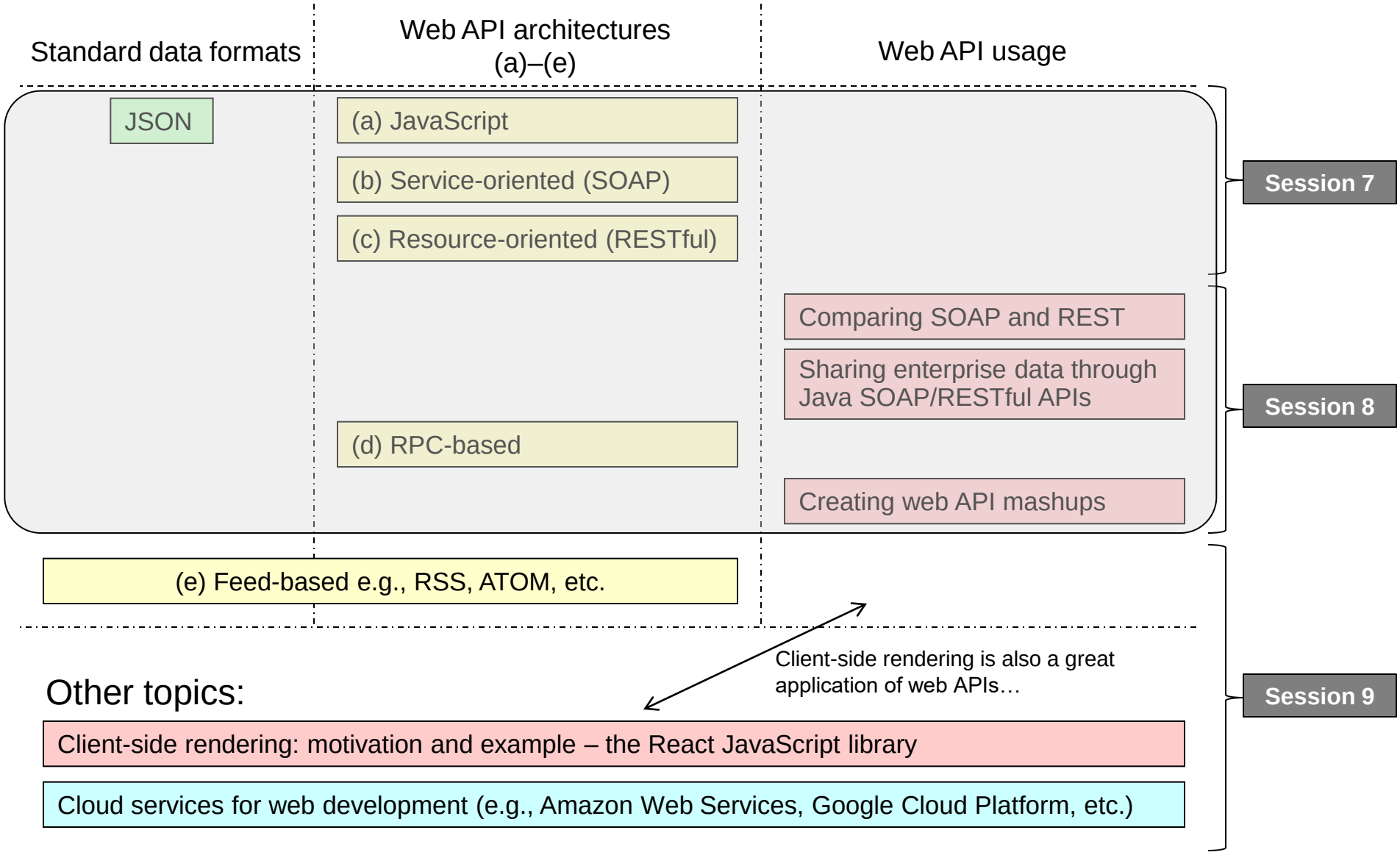
ICOM 6034
Website engineering

Dr. Roy Ho

Department of Computer Science, HKU

Session 9: Web API protocols (Part III), client-side rendering and cloud services for website development

Organization of Part 3 (Sessions 7-9)



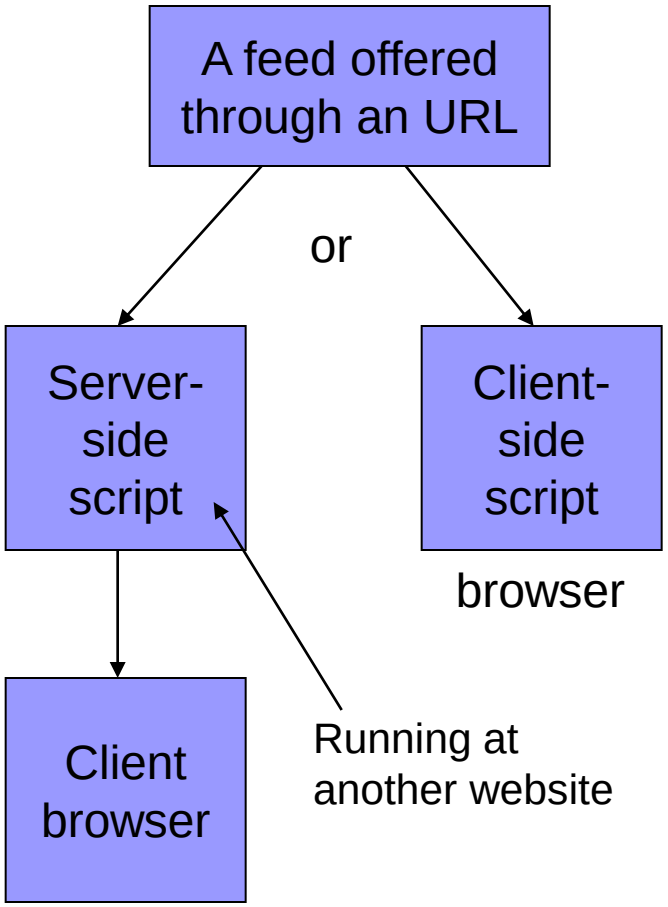
Session objectives

- Feed-based web APIs
- Summary of web API protocols
- Client-side rendering
  - Motivations
  - Introduction to React
- Cloud services for web development
  - Introduction to cloud computing
  - Popular cloud services:
    - Amazon Web Services (PaaS + IaaS)
    - Google App Engine (PaaS)
  - Security issues
- Summary of Part 3 of the course
- Demo: Google App Engine

Web API protocol (e): Feed-based web APIs

Feed-based web APIs

- API consumers retrieve XML data, structured as “feeds”, produced by API providers
- Common feed formats:
  - RSS feeds
  - ATOM feeds
- Each feed entry usually includes:
  - A unique ID
  - Timestamp
  - Title
  - The data content
  - ……
- Can be used for modeling almost any kinds of time-indexed data entries



Introduction to RSS

- RSS:
  - Really Simple Syndication
  - Rich Site Summary
- RSS feeds:
  - Commonly used to structure updated lists of news headlines and articles that users can subscribe to
  - Can also be used to model any collections of data items, especially those indexed by time
- Easily parsed by programs (XML)
- Read by human through software application such as **feed reader** or **aggregator.**
- This architecture allows the API clients to gather all the updated data items from multiple sources
- The updates of feeds can be "pulled" by the clients periodically– emulating a "push" effect

```
<rss version="2.0">
<channel>
<title>University of Rochester : Humanities and Social Science Releases</title>
<description>Updated news from University Public Relations</description>
<link>http://www.rochester.edu/news/search.php?cat=humansocial</link>
<copyright>Copyright 2015 : University of Rochester</copyright>
<image>
    <title>University of Rochester News</title>
    <url>http://www.rochester.edu/news/prgraphic.gif</url>
    <link>http://www.rochester.edu/news/</link>
</image>
<item>
    <title>University Celebrates 20th Anniversary of Annual Viennese Ball</title>
    <description>The Viennese Ball, one of the University of Rochester&amp;#39;s most popular annual events, will be held from 9 p.m. to midnight on Saturday, Nov. 12, in Wilson Commons on the University&amp;#39;s River Campus.</description>
    <link>http://www.rochester.edu/news/show.php?id=2312</link>
    <pubDate>Fri, 28 Oct 2015 00:00:00 EDT</pubDate>
</item>
<item>
    <title>Talk on China&amp;#39;s role in global business by David McHardy Reid</title>
    <description>International business expert and strategist David McHardy Reid will discuss China&amp;#39;s past 25 years of economic progress and how it may affect the future of global business at 2 p.m. Sunday, Nov. 13, in the Welles-Brown Room of Rush Rhees Library.</description>
    <link>http://www.rochester.edu/news/show.php?id=2306</link>
    <pubDate>Thu, 27 Oct 2015 00:00:00 EDT</pubDate>
</item>
</channel>
</rss>
```

Consuming and creating RSS feeds

- "Parsing" of RSS feeds is supported in many client-/server-side frameworks and libraries.
- Most CMS generate feeds for the managed contents automatically.
- Apart from using a CMS, the most common way to create an RSS feed is to read data from a backend database, then construct the feed according to the RSS format
  - E.g., you can use PHP (or other server-side language) to generate your RSS feed.
  - When a new item is added to the database, the script can be executed again to update the feed.

Generating an RSS feed

(an example using PHP and MySQL)

1) Assuming that your PHP program has connected to the MySQL database

2) Set the HTTP header

   Use the header() function to specify the content type of the feed (which is XML):

```
<? header('Content-type: text/xml'); ?>
```

3) Declare the RSS version (2.0) with the <rss> tag

4) Use the <channel> tag to begin your feed

5) Set up the feed's metadata using the <title>, <description>, and <link> tags.

(3)-(5) can be sent to the client simply by using "echo" in PHP.

Generating an RSS feed

6) Retrieve the records from the database

```
<?
$getItems = "SELECT refno, title, body, UNIX_TIMESTAMP(publication_date) AS pubDate
       FROM news ORDER BY publication_date DESC LIMIT 10";

$result = mysql_query($getItems);
$num_rows = mysql_num_rows($result);

if($num_rows !=0) {
       while ( $myrow = mysql_fetch_array($result) ) {
           $refno = $myrow["refno"];
           $title = $myrow["title"];
           $body=strip_tags($myrow['body']);
           $publication_date = strftime("%a, %d %b %Y %T
                                %Z",($myrow["pubDate"]));
           // to be continued in the next slide...
?>
```

This query retrieves the reference number, title, body text, and publication date for the most recent 10 records from the “news” table and sorts them in descending order by date.

Generating an RSS feed

7) Build the feed entries one by one
Format the title, description, and link for each item.

```
// code continued from last slide...
<item>
  <title><?print htmlspecialchars($title, ENT_QUOTES);?></title>
  <description><?print htmlspecialchars($body, ENT_QUOTES);?></description>
  <link>http://testingsite.com/news/showdetail.php?id=<?print $refno;?></link>
  <pubDate><?print $publication_date;?></pubDate>
</item>
<?
        }
    }
    else { ?>
       <p>No recent Press Releases found.</p>
<?  } ?>
```

Then end the feed with </channel> and </rss>.

ATOM feeds

- An RFC (request for comments) by IETF
- Structure of ATOM feeds is very similar to RSS - also based on XML
- RSS may contain either plain text or HTML as content, but no way to indicate which of the two is used.
- Atom allows plain text, XML, XHTML, HTML, and supports links to external content such as video, audio, documents, etc.

A simple atom feed

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

 <title>Dan's Blog</title>
 <link href="http://netzooid.com/blog/"/>
 <updated>2017-11-07T18:30:02Z</updated>
 <author>
   <name>Dan Diephouse</name>
 </author>
 <id>urn:uuid:60a76c80-d399-11d9-b91C-0003939e0af6</id>

 <entry>
   <title>Building services with AtomPub</title>
   <link href="http://netzooid.com/blog/atompub_services"/>
   <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
   <updated>2017-11-07T18:30:02Z</updated>
   <content>
     … (you must have content or a summary)
   </content>
 </entry>

</feed>
```

- **Feed**
  - ID
  - Author
  - Link
  - Title
  - Updated
  - *
  - **Entry**
    - ID
    - Updated
    - Link
    - Summary
    - Content
    - *

Summary of web API protocols

Summary of web API protocols

- (a) JavaScript
  - A JS library is given to the API clients, e.g., the Google Maps library
  - Good for small-scale data sharing (e.g., just to share a widget)
  - JS - easy to use at client-side, but server-side usage is limited
  - No standard to follow on how data (or the web API) is actually modeled
    - Limited compatibility and extensibility

- (b) Service-oriented (or SOAP) APIs
  - Good programming support in the enterprise application domain
  - Flexibility in defining the API interface, interaction patterns between service and consumers – not limited to request-response pairs
  - It may be more straightforward to model a traditional enterprise application (e.g., the stock broker application that we introduced in Session 7) as a SOAP API
  - Disadvantages: opposite to the advantages of the resource-oriented approach – see below.

- (c) Resource-oriented (or RESTful) APIs
  - Great scalability (as proven by the web), interoperability and performance by using the existing, highly-optimized web servers and protocol (HTTP)
  - A RESTful API can be made by using the URL routing facilities of many server-side frameworks (e.g., Laravel, Rails, etc.)
    - No need to use a custom protocol (e.g., SOAP) or formulate the WSDL contracts
  - The most-advocated web API architecture in the community
  - Need to model everything as a resource and have a coherent URL space
    - May not be a trivial task if the backend system (data source and application logic) isn't modeled in that way

Summary of web API protocols

- (d) RPC-based
  - Simplified versions of service-oriented APIs without WSDL
    - Like function calls => easy to use; great for ad-hoc or small-scale data sharing
    - Share similar weakness with service-oriented approach when compared to resource-oriented approach
    - Open-source libraries for different languages are available
- (e) Feed-based
  - Good for sharing data modeled as an array of (time-indexed) entries
  - Easy to implement
  - No pre-defined pattern on how the clients and servers interact
    - This approach is mainly designed for data dissemination => one-way communication; “interactions” between API providers and consumers are limited
- Each architecture has its pros and cons; the choice should be based on project needs and whether your desired server-/client-side frameworks support that protocol or not.

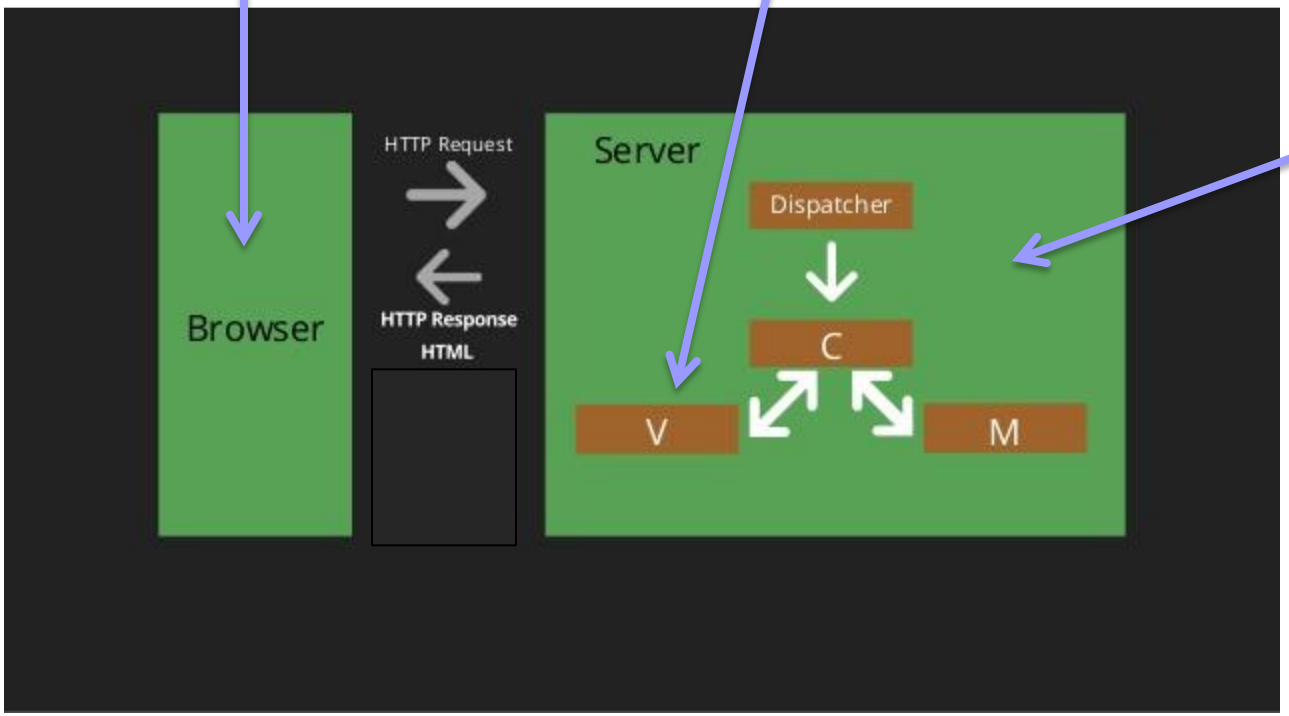
Client-side rendering: the motivations

In the beginning, we only had server-side rendering…

- Probably since the introduction of "dynamic data" on the web, MVC (at the server side) has been proven as an effective way to structure server-side code and make it extensible and maintainable.

Clients simply display the generated HTML (plus CSS+JS)

The rendering of web pages (i.e., the generation of HTML) is done by the server.

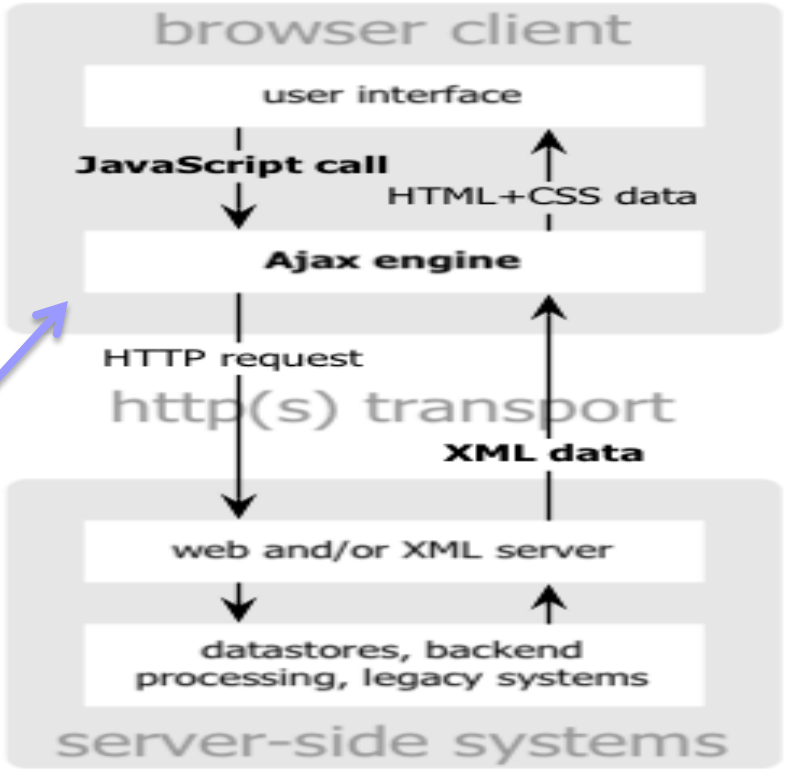


M-V-C promotes **separation** of concerns so that application logic ("C"), data management ("M") and presentation ("V") can be made independent from each other, therefore facilitating application extensibility and maintainability.

But: what happened after AJAX appeared?

- In order to better simulate a “desktop-like” user experience, AJAX has been widely used.
- Updates of web page components are fetched from the server, without refreshing the entire page.
- When these “Web 2.0” websites/apps have become popular, the communication between servers and clients (and the **rendering** of web pages) have been gradually shifted to a two-stage pattern:
  - 1. The display of the initial web page (**rendered by the server-side MVC**)
  - 2. The AJAX requests and responses for subsequent updates of web page components

When the data arrives, the AJAX code identifies the webpage components and updates their content in HTML => i.e., the client-side JS has started to share the work of **(re-)rendering** the HTML.



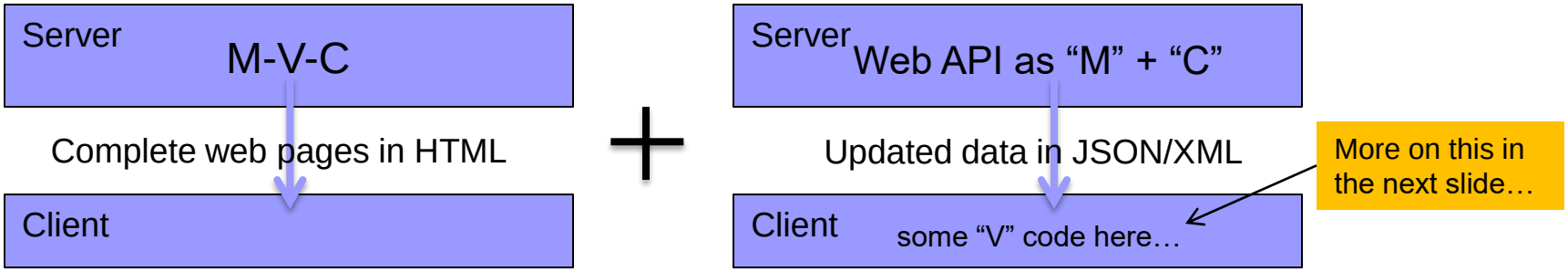
Ajax web application model

Issue #1: server's role is "overloaded"

- In practice, the 2nd stage of client-server communication (for retrieving updated data) is mostly done through web APIs (e.g., RESTful/SOAP endpoints).

- Result: developers not only need to implement the server-side rendering code (for displaying the initial web page), but also a web API for providing data for subsequent updates.

- The server now has two roles:
  - 1) it itself is a complete Model-View-Controller system (for all "initial-page" requests); and:
  - 2) it will also act as a web API (the "Model" + "Controller") for providing "raw data" to client for AJAX updates.



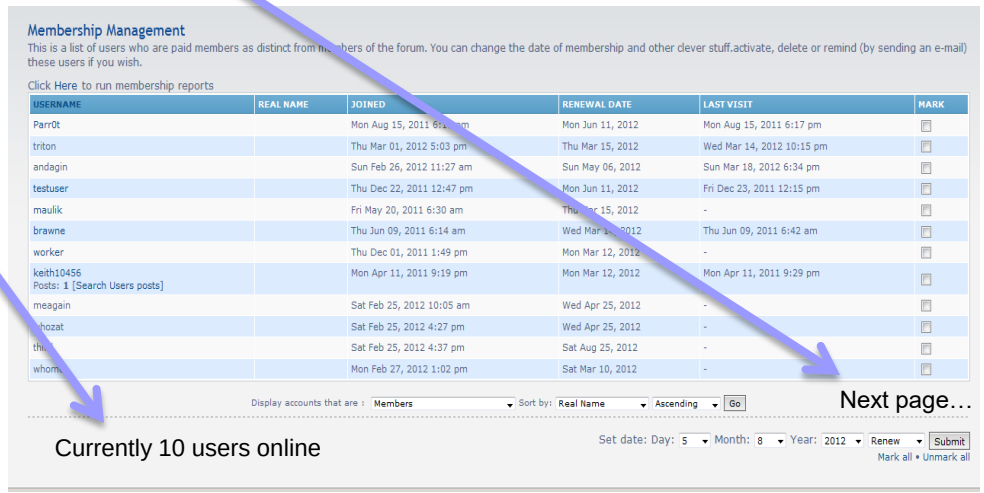
- Higher implementation cost, less structured code
  - Result: less maintainability and extensibility, therefore **compromising the original goal of using MVC**

Issue #2: duplication of rendering logic

- Suppose we would like to list all members of an online forum using AJAX pagination – each time you click “Next page…” an AJAX request is sent to the server and HTML is retrieved back to update the table. This is fine.

- But: what if we need to update another counter: number of users online at the moment? Two “solutions”:

1. The AJAX response is restructured to contain both the list of users and the new counter. The AJAX code would need to decode the response (“raw data”), generate/render two HTML segments, identify the related HTML elements, and update the table and counter accordingly.
2. Send another AJAX request for the new counter.



- (2) is not preferred since the number of such requests can grow (e.g., even more content elements need to be updated), which can increase the server’s loading.

- For (1), it would mean that the rendering logic (for translating raw data to HTML) has to be implemented both at the server side (for the initial page) and client side (in the AJAX callback).
  - Logic duplication compromises the original goal of MVC – separation of concerns
  - E.g., if the logic is changed, both server & client code would need to be modified.

On moving the MVC/rendering to client-side

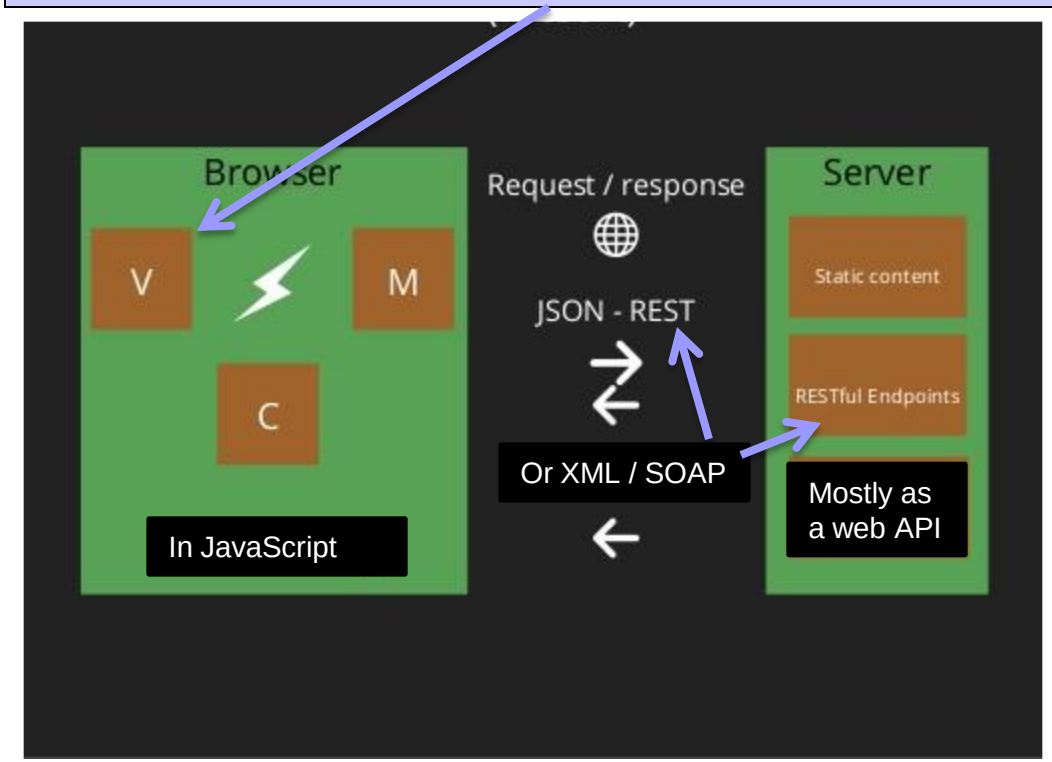
Other motivations to client-side rendering:

- Reduced server load; better server scalability
- Web apps (i.e., implemented in HTML/CSS/JS) packaged as native mobile apps
  - The "model" can obtain data from web API (when online) or local storage (when offline)
- Single-page applications (SPA) where updates of web page elements is common
  - E.g., Gmail, Facebook, etc.

Key enabling factors:

- Web APIs! (i.e., what we have covered in Sessions 7-8)
- More powerful clients (smartphones, tablets, browsers, etc.)
- HTML5: web/offline storage, web sockets, etc.

Some frameworks/libraries (e.g., React) implement only the "view" component at the client-side (for rendering), while the "model" and the "controller" functions are implemented at the server-side as part of the web API.



- However, client-side rendering isn't problem-free:
  - Browser compatibility (older IEs, Opera Mini, etc.)
  - Poor client experiences in slow devices
  - SEO becomes more difficult (solutions available in some JS frameworks)

Client-side JavaScript frameworks/libraries for client-side rendering

- React by Facebook (mainly for the "View" component)
- Angular by Google
- Vue.js
- Ember.js
- Backbone.js… and many others

- See TodoMVC.com for more examples
  - This website implements the same Todo application by using different client-side "**MV* frameworks**". Developers can view the source code and decide which framework to use.

Some of these frameworks are not exactly MVC but **variants** of it. For examples:

- Backbone.js is a "model-view-collections" framework.
- React is the "view" component; the server (web API) performs the "M" and the "C" parts.
- Angular is a component-based framework.

Other variants of MVC include MVT (model-view-template), MVP (model-view-presenter), MVVM (model-view-viewmodel), etc., and a few others. These are generally called "MV*" patterns.

No matter how a pattern is named, they all have one thing in common – clean **separation of concerns**. See their websites for how separations are done in their designs.

Introduction to React JS

An example of client-side rendering

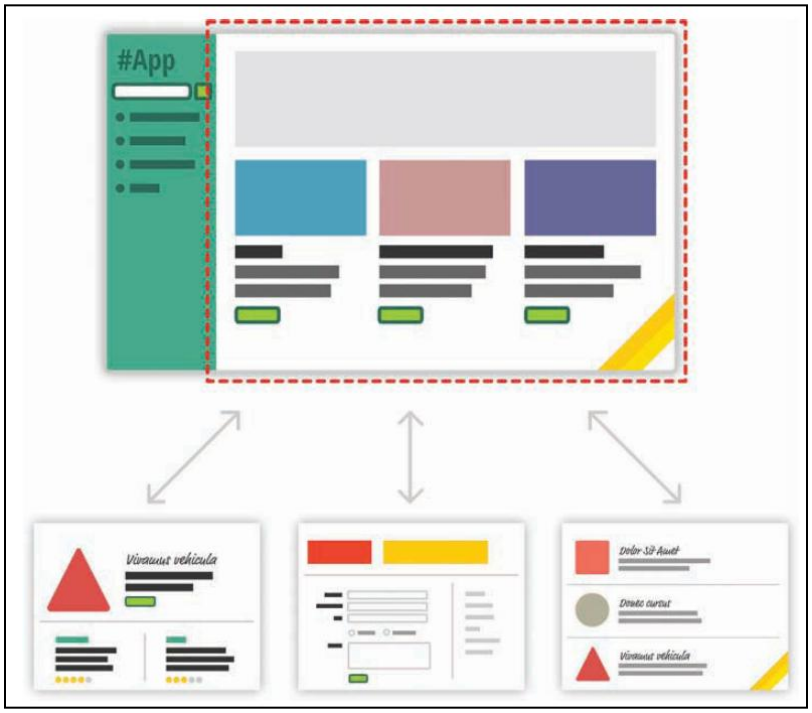
Notes: the program code in this section will not be covered in the examination.

Some examples were taken from K. Chinnathambi, “Learning React,” Addison-Wesley, 2018, and w3schools.com’s tutorial.

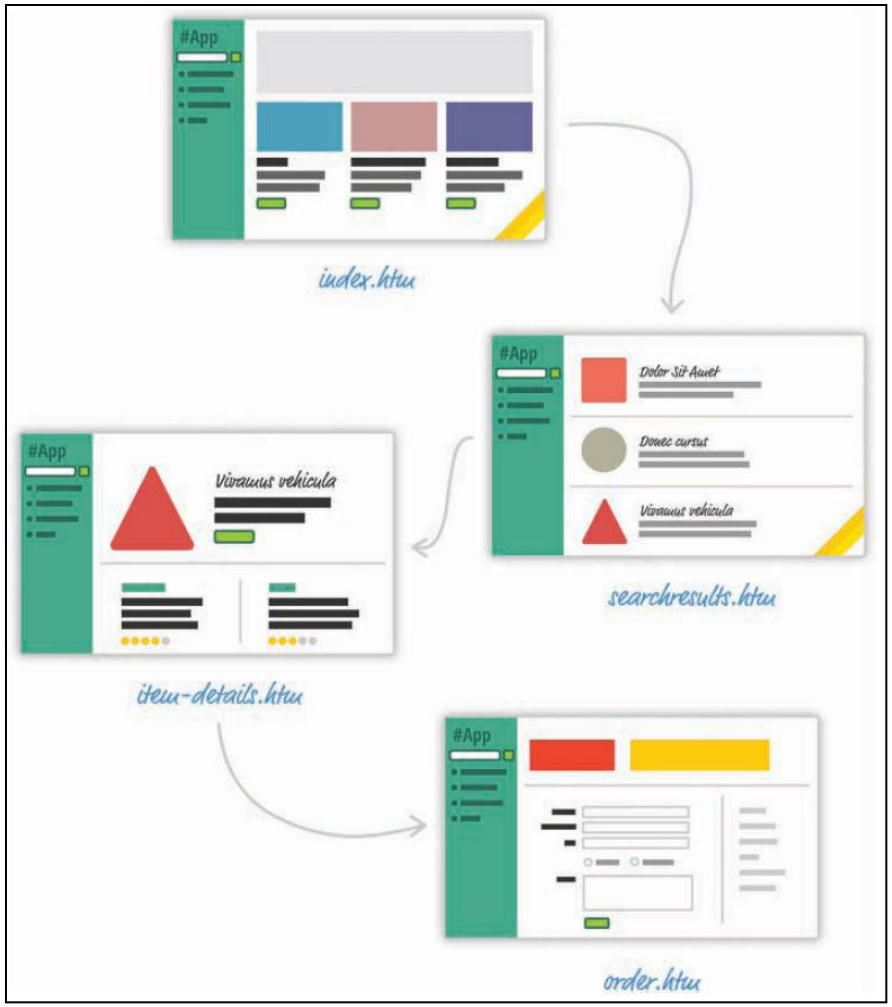
React JS library (or simply React)

- Developed by Facebook
- For implementing the "**view**" component of an MVC application
  - The server (i.e., the web API) would perform the tasks of the "model" and the "controller".
- For easily building single-page applications

Single-page application with client-side rendering:

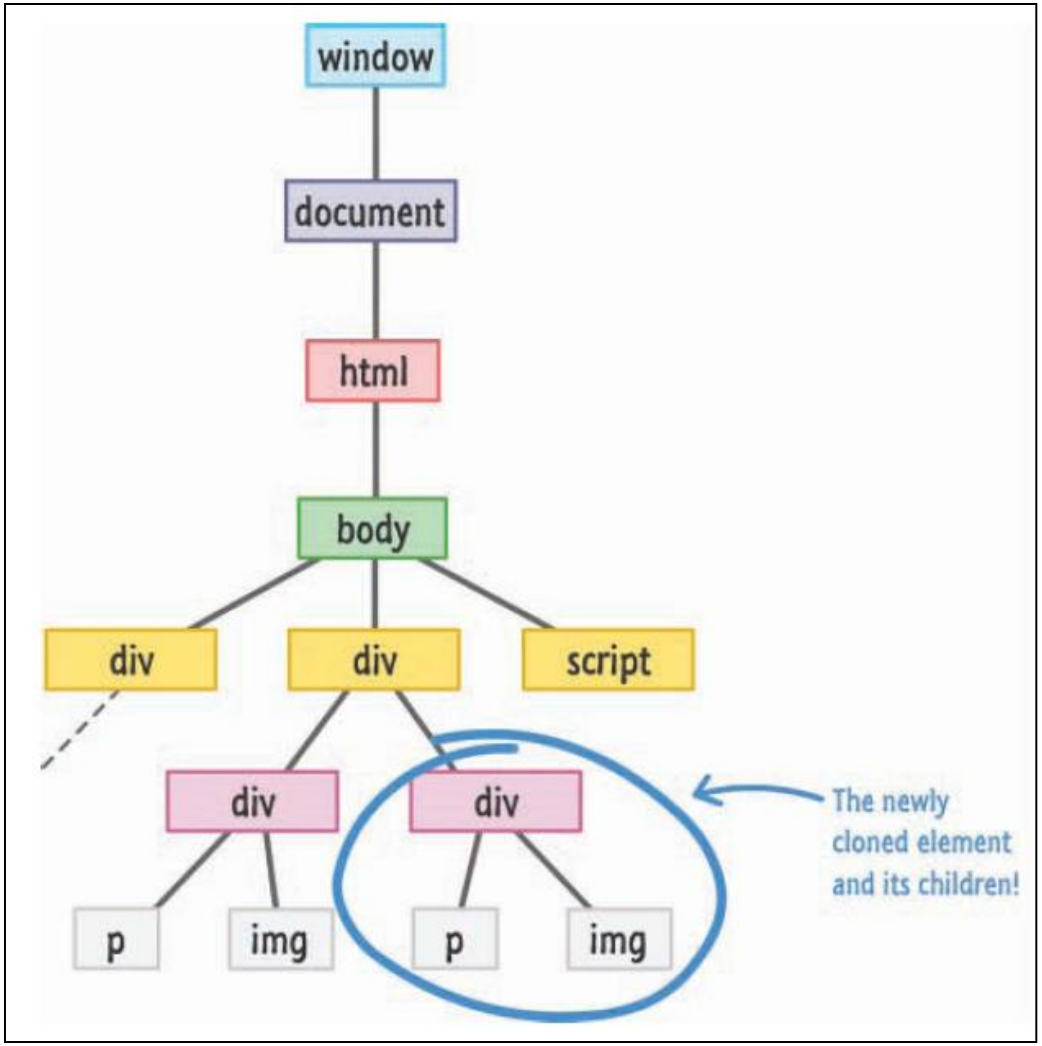


Traditional, page-based design with server-side rendering:



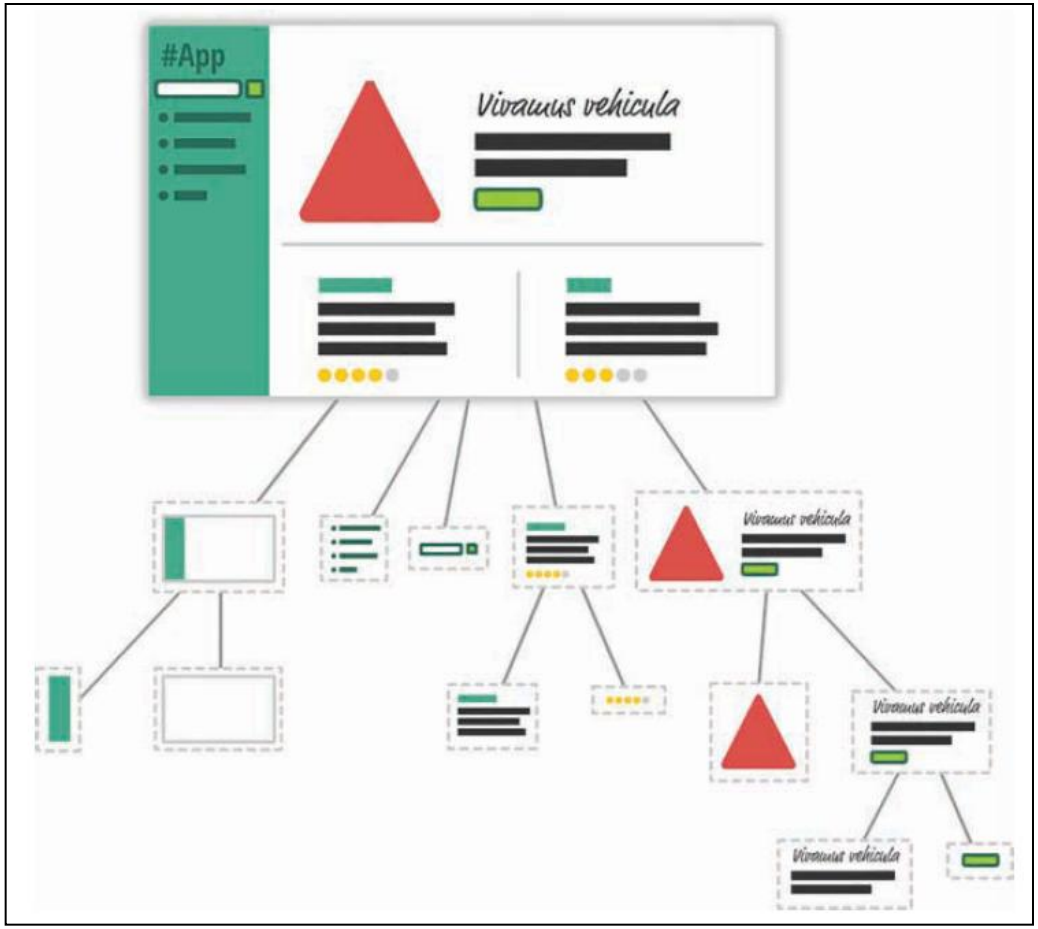
Virtual DOM

- Single-page applications usually have a large number of small updates to the UI elements.
- Traditionally, updates of these elements require manipulations of the browser DOM, which is extremely slow.
- React manages an in-memory "**virtual DOM**" for hosting all intermediate/temporary updates, while only those essential, final updates are made to the real DOM.
  - Manipulations of the virtual DOM is *much faster*
  - => Much faster execution of the entire application



Component-based view designs

- Instead of treating the visual elements in a page as one “monolithic chunk” (e.g., as in plain PHP), React encourages breaking down your visual elements into smaller, more manageable and **reusable components**.
  - Components = building blocks of a React application.
- Each component has its own embedded:
  - Content (or “states”)
  - CSS
  - JavaScript (i.e., behaviors)
- Components can be “nested”, i.e., a component can contain a number of smaller components.
- The component-based design is actually another way to perform **separations**, while a single component can also be easily **re-used** in different parts of your application.



Hello world

React core libraries

The **Babel** library converts **JSX** to JavaScript

An empty to host the rendered “Hello World!”

A React **component** called “Hello”

This function will be executed when this component is **rendered**

```
<!DOCTYPE html>
<html>
<script src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
<body>

<div id="mydiv"></div>

<script type="text/babel">
    class Hello extends React.Component {
        render() {
            return <h1>Hello World!</h1>
        }
    }

    ReactDOM.render(<Hello />, document.getElementById('mydiv'))
</script>

</body>
</html>
```

This script will first be translated (JSX => JS) by Babel

To render the “Hello” component inside the mydiv element

React code is largely written in **JSX**, an extension to JavaScript which allows for easy handling of HTML inside JS code.

Nested components to form a bigger component

React JS:

```
class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}

ReactDOM.render(<Garage />, document.getElementById('root'));
```

HTML:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>
  <body>

    <div id="root"></div>

  </body>
</html>
```

Result:

Who lives in my Garage?

I am a Car!

When a component is no longer needed, it could be hidden (by using CSS) or **unmount** from the DOM (beyond the scope of our discussion).

React props

React JS:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car brand="Ford" />
      </div>
    );
  }
}

ReactDOM.render(<Garage />, document.getElementById('root'));
```

Prop as a parameter to be passed to the component being rendered.

HTML:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>
  <body>

    <div id="root"></div>

  </body>
</html>
```

Result:

Who lives in my Garage?

I am a Ford!

Encapsulating styles in a component

```
class MyHeader extends React.Component {
  render() {
    const mystyle = {
      color: "white",
      backgroundColor: "DodgerBlue",
      padding: "10px",
      fontFamily: "Arial"
    };
    return (
      <div>
      <h1 style={mystyle}>Hello Style!</h1>
      <p>Add a little style!</p>
      </div>
    );
  }
}
```

A style object

Applying a style for an element

By encapsulating the styles inside a component, the component could be easily **reusable** – just like a black box.

Different sets of styles can be chosen and applied (e.g., for different devices) by using props.

Note the clean separation between presentation details and HTML is still maintained.

Automatic re-rendering of state/components

In HTML:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>
  <body>

    <div id="root"></div>

  </body>
</html>
```

In JavaScript:

```jsx
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { username: '' };
  }
  myChangeHandler = (event) => {
    this.setState({username: event.target.value});
  }
  render() {
    return (
      <form>
      <h1>Hello {this.state.username}</h1>
      <p>Enter your name:</p>
      <input
        type='text'
        onChange={this.myChangeHandler}
      />
      </form>
    );
  }
}

ReactDOM.render(<MyForm />, document.getElementById('root'));
```

Whenever the state is updated, the entire component would be re-rendered.

If this state is linked with a result returned from a web API, the component will also be updated when new data is arrived from the server.

Much more convenient than the traditional way of handling AJAX data (e.g., first identify the web page component to be updated, and then update its value through DOM, etc.).

https://www.w3schools.com/REACT/showreact.asp?filename=demo2_react_forms_handling

Choosing between server- and client-side rendering

- In general, the following websites/applications might better be implemented with server-side rendering:
  - Traditional, "page-based" websites which do not have much dynamic data within a page
  - Those which have complicated application logic (i.e., slow to run in the client-side)
  - Those which involve computation based on a large amount of backend data => too expensive to fetch that data to the client side
- Client-side rendering is best used to implement webapps which can run offline, and single-page applications (SPAs) which have lots of (small) updates and user interaction within each page
  - E.g., Gmail, Facebook, etc.
- For those websites which have both of the above properties, one can consider to use both server- and client-side rendering:
  - E.g., use server-side rendering to prepare an initial page, part of which contains the elements which are created (and then managed) by the client-side rendering.
    - I.e. the client-side rendering is part of the "V" produced by the server-side rendering.
  - Still important to avoid duplication of code – the original goal of "separation of concerns".

Cloud computing for website development

“Cloud computing” - definition

- Can be different things to different people
- A generally agreed usage of the term:
  - A style of computing in which computing facilities are provided "as a service" through Internet to the clients,

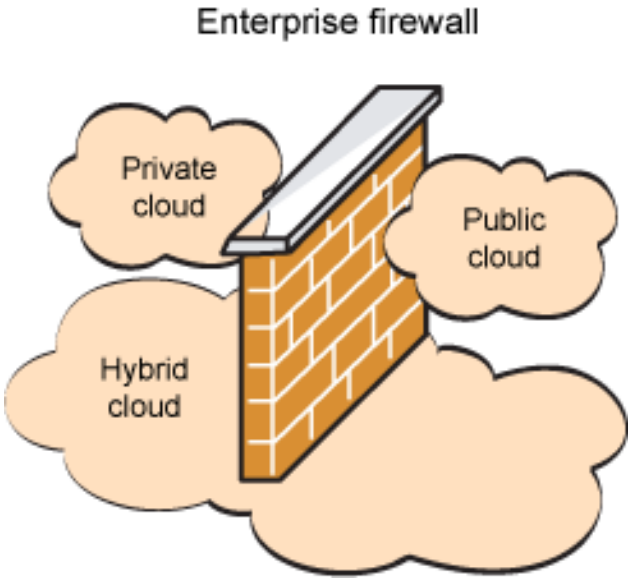
    which:-
  - Enables convenient, on-demand access to a shared pool of virtualized computing facilities (e.g., networks, CPU/servers, storage, applications, etc.) that can be rapidly scalable, provisioned and released with minimal management effort.

Some common properties

- On-demand => massive in scale and rapidly scalable
- Broad network access, location independence, and geographic distribution
  - Many cloud providers provide local access in different regions
- Highly reliable => users can have replicas of applications
- Measured usage and service-level agreements (SLA) for leasing
- Advanced security technologies
- A **cloud** can further be characterized by the deployment models and the service models (SaaS, PaaS, IaaS).

Cloud deployment models

- Private cloud
  - Enterprise owned or leased
- Public cloud
  - Mega-scale infrastructures, sold to the public
  - E.g., Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), IBM Cloud, Oracle Cloud, Alibaba Cloud, etc.
- Hybrid cloud (or “multi-cloud”)
  - Two or more clouds
  - Can consists of multiple public clouds, or public cloud + private cloud



Cloud service models or service layers

Cloud Market Types	Types of Offerings	Examples	
Software-as-a-Service	• Rich Internet application web sites • Application as Web Sites • Collaboration and email • Office Productivity • Client apps that connect to services in the cloud	• Flikr • Myspace.com • Cisco WebEx office • Gmail • IBM Bluehouse	SaaS
App-components-as-a-Service	• APIs for specific service access for integration • Web-based software service than can combine to create new services, as in a mashup	• Amazon Flexible Payments Service and DevPay • Salesforce.com's AppExchange • Yahoo! Maps API • Google Calendar API • zembly	
Platform-as-a-Service	• Development-platform-as-a-service • Database • Message Queue • App Servicer • Blob or object data stores	• Google App Engine and BigTable • Microsoft SQL Server Data Services • Engine Yard • Salesforce.com's Force.com	PaaS
Infrastructure-as-a-Service	• Virtual servers • Logical disks • VLAN networks • Systems Management	• Akamai • Amazon EC2 • CohesiveFT • Mosso (from Rackspace) • Joyent Accelerators • Nirvanix Storage Delivery Network	IaaS
Physical Infrastructure	• Managed Hosting • Collocation • Internet Service Provider • Unmanaged hosting	• GoDaddy.com • Rackspace • Savvis	

Sessions 7 & 8

This lecture

Adapted from Forrester Research Taxonomy

Software as a Service (SaaS)

- Examples: Gmail, Google Docs, Facebook……
- Cloud provider offers software (as a service) to end users
- Users may buy subscriptions to software or components
- The software is run on the cloud provider's platform
- "App-component-as-a-service" is a variant of SaaS:
  - Only components (instead of complete software) are offered
  - Not for direct use, but for data integration, e.g., for mashups
  - Mostly available through web APIs

Platform as a Service (PaaS)

- Cloud provider offers a development platform (including OS, database engines, development/runtime libraries, etc.) on which developers implement applications
- The developed application is run on the cloud provider's platform.
- Example: Google App Engine

Infrastructure as a Service (IaaS)

- Cloud provider provides virtualized, IT infrastructure to clients
  - E.g., CPU cycles (or virtual machines), virtualized storage space, virtual private networks, etc.
    - These resources can grow and shrink on demand, e.g., from one virtual machine instance to thousands
- For virtual machines, users can deploy their own software environment (i.e. virtual machine images incorporating OS and libraries)
- Users have complete control on the system performance and scalability
- Example: Amazon EC2

Management matrix

= Managed for You	Stand-alone Servers	IaaS	PaaS	SaaS
Applications	✗	✗	✗	✓
Runtimes	✗	✗	✓	✓
Database	✗	✗	✓	✓
Operating System	✗	✗	✓	✓
Virtualization	✗	✓	✓	✓
Server	✗	✓	✓	✓
Storage	✗	✓	✓	✓
Networking	✗	✓	✓	✓

It is trivial that the more components being managed by cloud provider, the less hassles users have to face, but what are the tradeoffs?
=> Flexibility, vendor lock-in, difficulties in outside-cloud service replication and data backup, etc.

Advantages of cloud computing

- Clouds vs. traditional web hosting
  - Cloud services are sold on-demand
  - They are elastic - a user can use as much or as little of a service as they want at anytime
    - "On-demand" + "elastic" => cost-effectiveness
  - Pay for what you need
  - Users do not have to purchase the "peak capacity" => reduced cost
- Managed computing environments
  - The service is fully managed by the provider at the desired level and according to users' needs – SaaS, PaaS, IaaS, etc.
  - Scalability, availability and performance are well taken care of
  - Tools for system monitoring, logging and security configurations are available
- Leverage others' core strengths
  - Managing a scalable, highly-available, secure and updated platform requires a full team of experienced engineers
    - Cloud vendors like Amazon and Google have decades of such experiences
  - Developers can focus on their application logic => shortened development time

Amazon Web Services

An example of a hybrid platform for PaaS and IaaS

Amazon Web Services (AWS)

- Advantages:
  - Pay-per-use model
    - You are only charged for the CPU time, network bandwidth and storage that you use.
  - Instant scalability
  - Most services are available through REST/SOAP APIs
  - Amazon's experiences
    - Amazon has been running (using) the platform for itself for 20+ years.

- Main components (can be used independently)
  - Elastic Compute Cloud (EC2)
  - Elastic Block Storage (disk storage volumes)
  - Simple Storage Service (S3) and SimpleDB
  - Simple Queue Service (SQS) – similar to JMS for Java EE
  - Relational Database Services (RDS) – "hosted MySQL"

IaaS

PaaS

We will introduce these 3 components

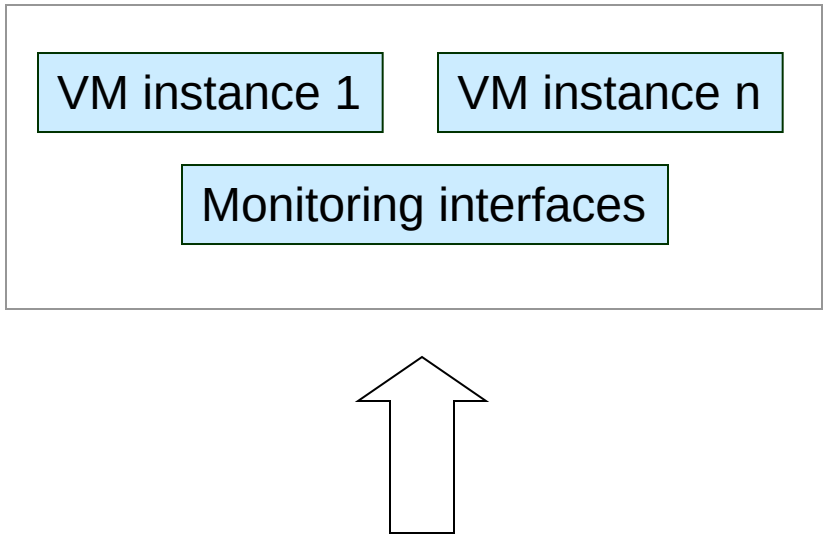
AWS has a number of other cloud services, e.g., DynamoDB (more scalable than SimpleDB, but less easy to use); see their website for detail.

Amazon EC2

- Elastic Compute Cloud
  - Rent virtual machines (VM) to run your software. Monitor and increase / decrease the number of VMs as demand changes
  - **Elastic**: you can grow or shrink the whole platform ***within minutes***
  - Administrative/root access to each VM
  - Flexible: choose your OS, software packages…
    - Redhat, Ubuntu, Windows, etc.
    - Small, large, extra large instances
  - Reliable: failover supports
  - Secure: e.g., web interface to configure firewall settings

- Example costs (check online for updated version):
  - CPU: small instance - US$0.10 per hour for Linux, $0.125 per hour for Windows (on server-grade hardware)
  - Bandwidth: in $0.10, out $0.17 per GB
  - Storage: $0.10 per GB-month, $0.10 per 1 million I/O requests

EC2



How to use:

- Create an Amazon Machine Image (AMI): applications, libraries, data and settings
- Upload AMI to Amazon
- Use Amazon EC2 web interface to configure security and network access
- Choose OS, start AMI instances, use them as regular machines
- Monitor & control through web interface or APIs

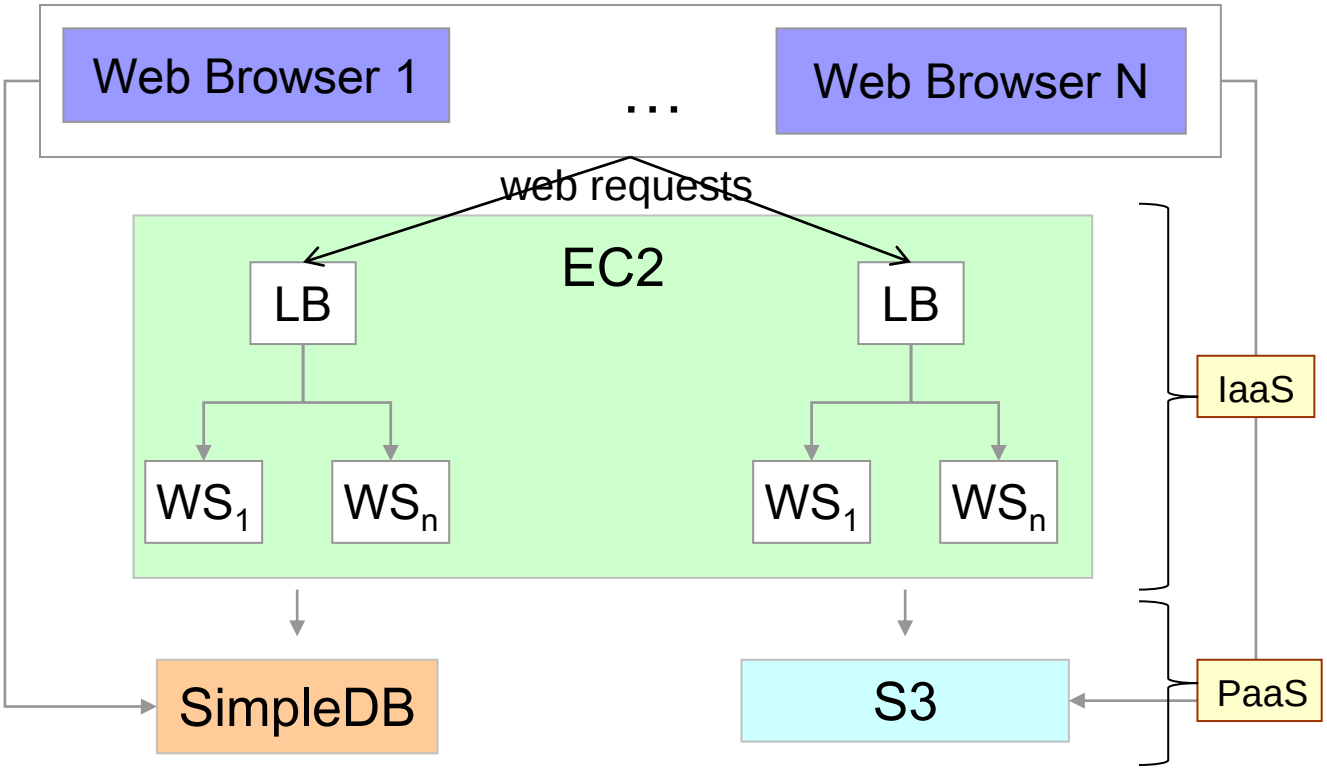
In addition to VM images, AWS can also run Docker containers and the Kubernetes container management tools in Amazon Elastic Container Service (ECS).

(Beyond our scope of discussion – feel free to contact me for more information).

Sample EC2 use cases

Web server farm:

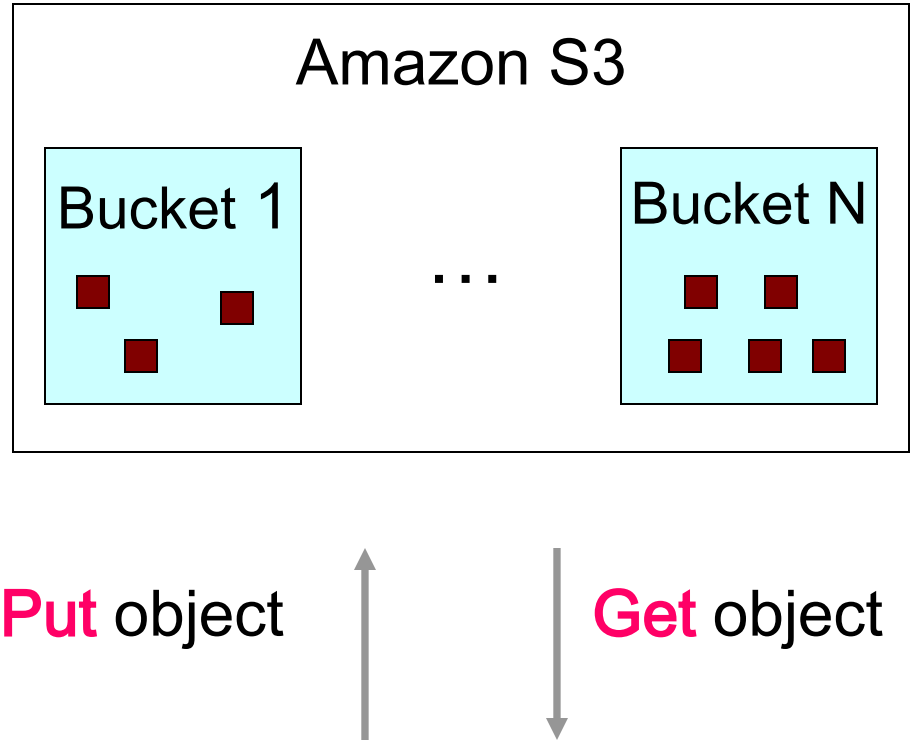
- All VMs are configured with the same web server.
- One or more additional VM act as the load balancer(s)
- The DNS server distributes requests between load balancers.
- The number of web server VMs can grow when traffic increases.



Another use case: **parallel computing:**

- All VMs share the same code (e.g., for generation of bank statements at night)
- Each VM operates on a subset of data.

Amazon S3



Idea:

Put/Get objects into “buckets” based on unique keys.

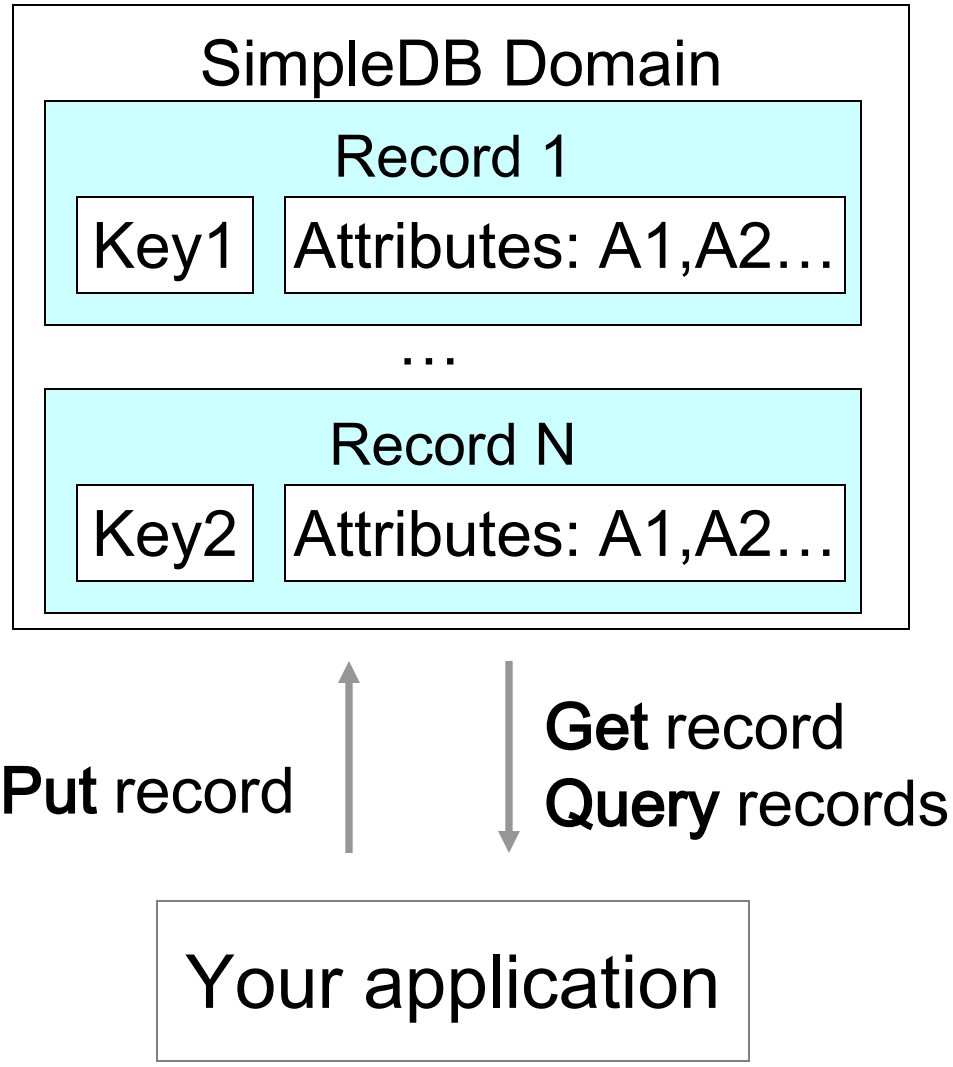
Main features:

- Public/Private access.
- Support for large objects.

Common use cases:

- Image / video storage - put your media files on S3 and then serve it
- Object serialization - objects created by your Java programs (running in EC2) can be serialized to S3 for later use

Amazon SimpleDB



A highly-scalable NoSQL data store

Idea:

Create flat database with auto-indexed tables.

Main Features:

- Each attribute is indexed.
  - Searching is fast
- Record structure is flexible.
- Queries are flexible.
- Supports sorting.

SimpleDB

Typical use cases:

- Store flat objects
  - Use SimpleDB as a storage for non-nested data (e.g., user profiles)
- Index and manage media/data files stored on S3
  - Metadata of each media/data file can be stored as attributes in SimpleDB

However, it doesn't support some advanced relational operators (UNION, etc), etc.

- Have to be implemented in applications' own logic, or use third-party libraries
- Or, use the Amazon's RDS (Relational Database Service), a tailored MySQL database for these operations.

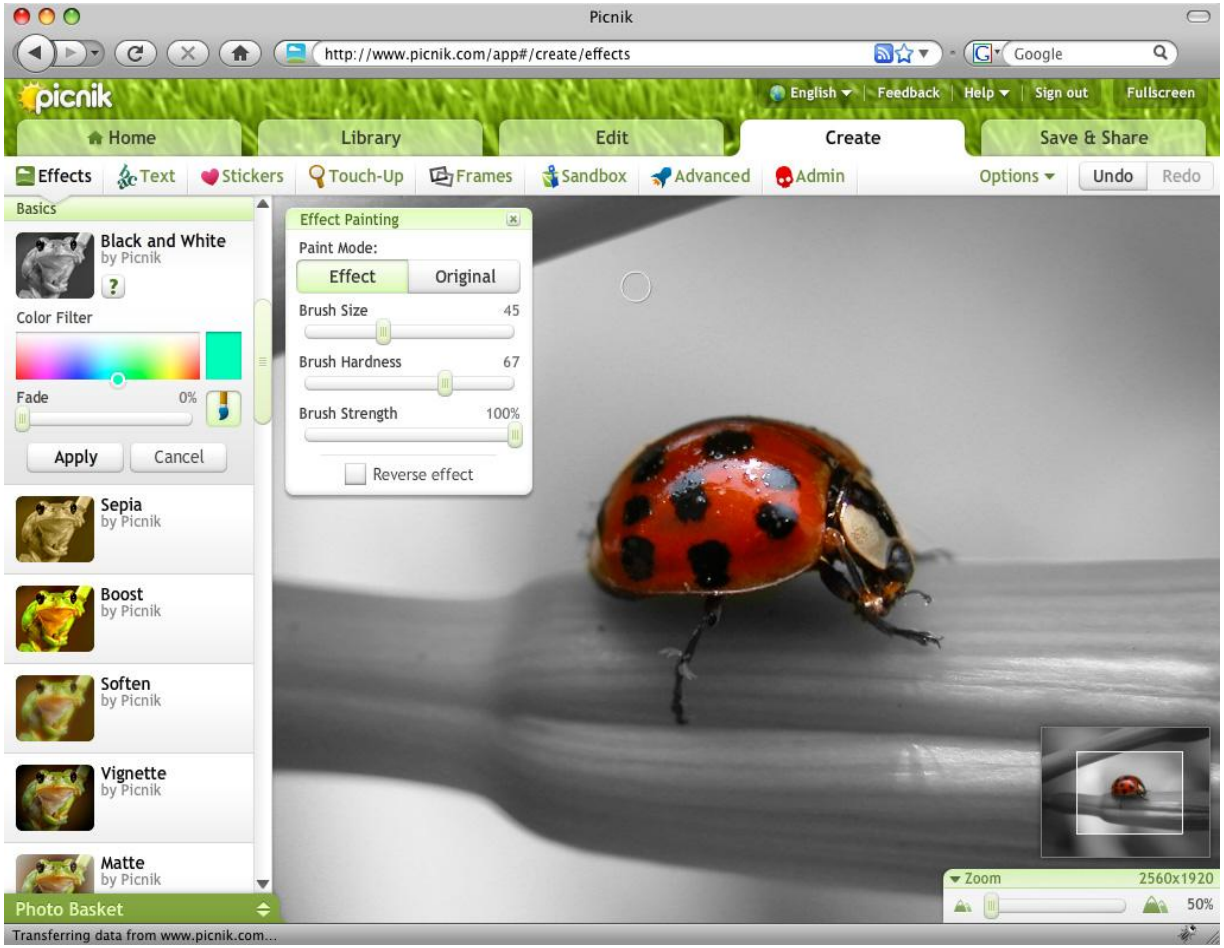
AWS case studies

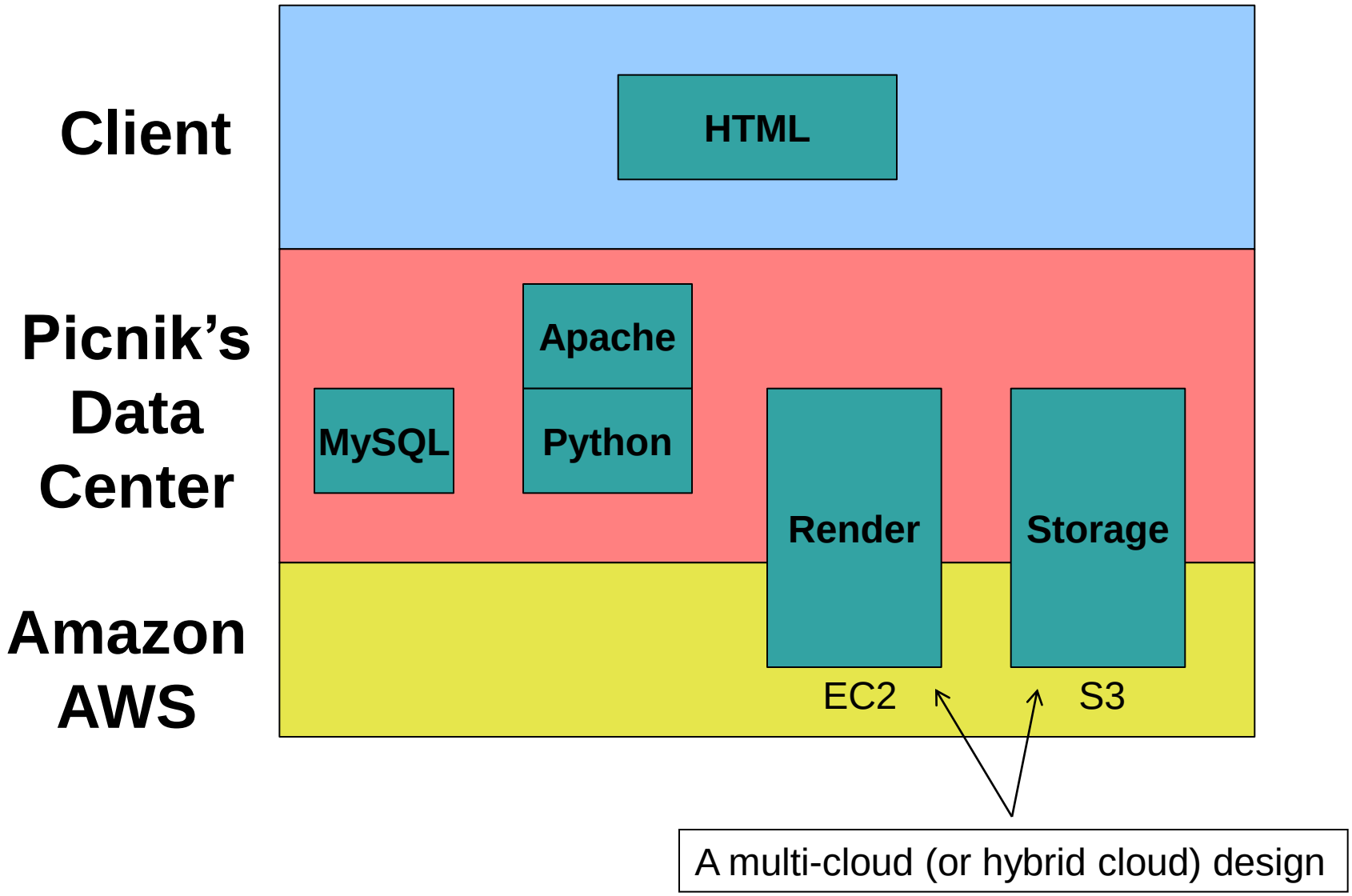
Case studies

- Instagram
  - Launched in 2010; on Amazon EC2
  - 25000 users on the 1st day; grew to 1M users in 3 months and 100M in 2 years; took thousands of VMs in EC2 to serve
- Netflix
  - Has been running 95% of its workload in AWS since 2013.
  - AWS has enabled Netflix to quickly deploy thousands of servers and terabytes of storage within minutes. Users can stream Netflix movies from anywhere in the world
- New York Times
  - Didn't collaborate with Amazon; NYT staff just used a credit card to pay online
    - Demonstrating how convenient the service is even for large-scale usage
  - Used EC2 and S3 to convert 15 million scanned news articles to PDF
  - Took 100 Linux VMs 24 hours - would have taken months on NYT's own computers
  - "It was cheap experimentation, and the learning curve isn't steep." – Derrick Gottfrid, NYT
- Nasdaq
  - Uses S3 to deliver historical stock and fund information
  - Millions of files recording stock prices over 10 minute segments
  - "The expenses of keeping all that data online [in Nasdaq servers] was too high." – Claude Courbois, Nasdaq VP

Picnik.com

- An online photo editing application
  - Now discontinued… the dev team was hired by Google; the photo editing functions are now available in some of Google's services.
- Huge demands on storage and computing power (e.g., for applying different effects to photos)

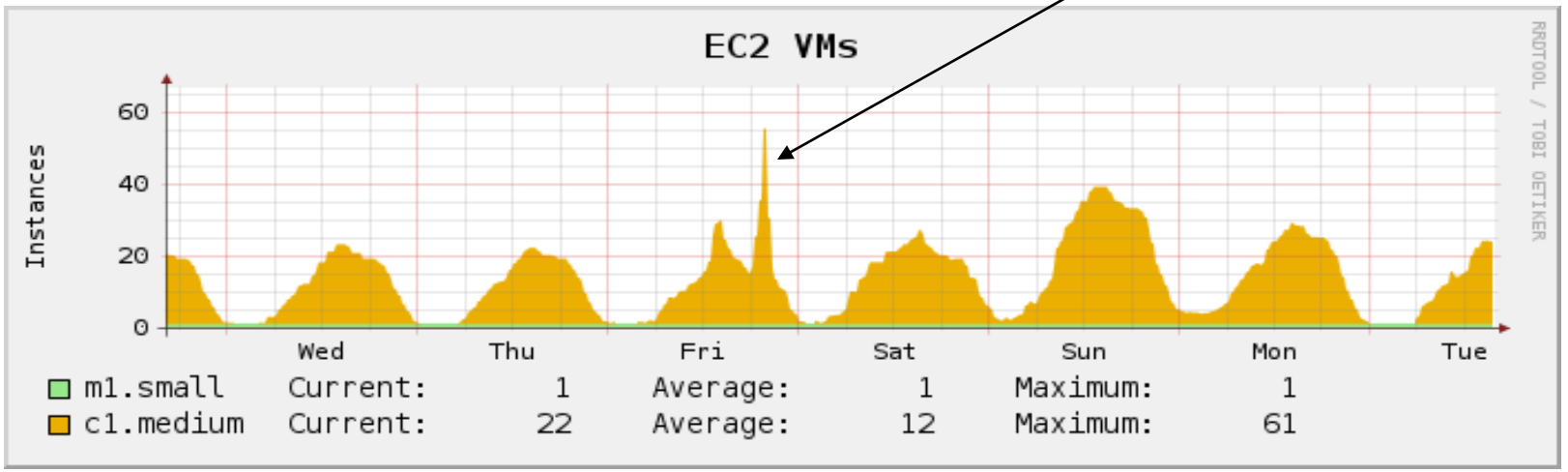



Client
HTML
Picnik's Data Center
Apache
MySQL
Python
Render
Storage
Amazon AWS
EC2
S3
A multi-cloud (or hybrid cloud) design

Rendering (computing at EC2)

- Manager process maintains enough idle “workers” (i.e., VMs)
- The number of workers can scale on-demand

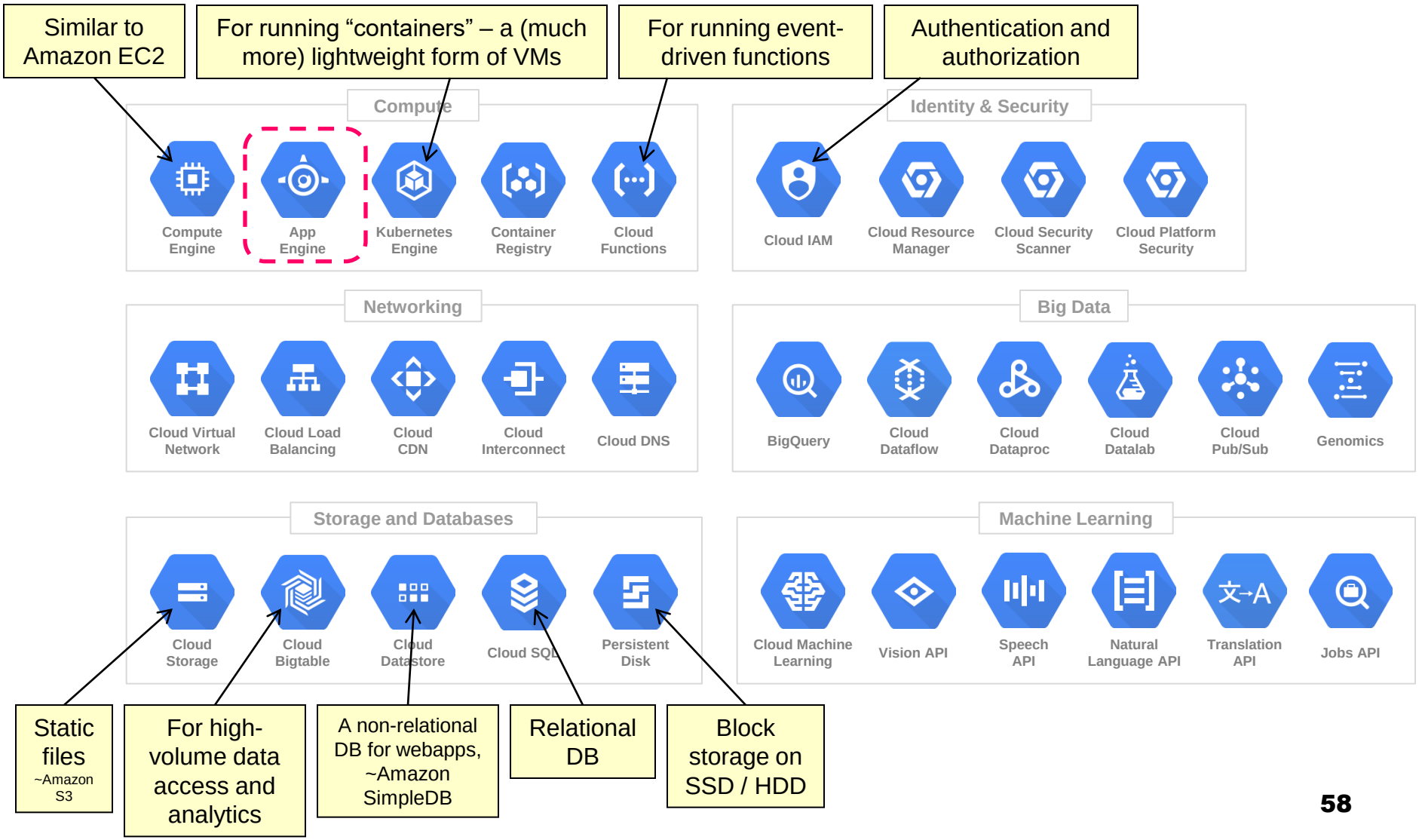
A peak that is otherwise impossible to handle without a platform as scalable as EC2



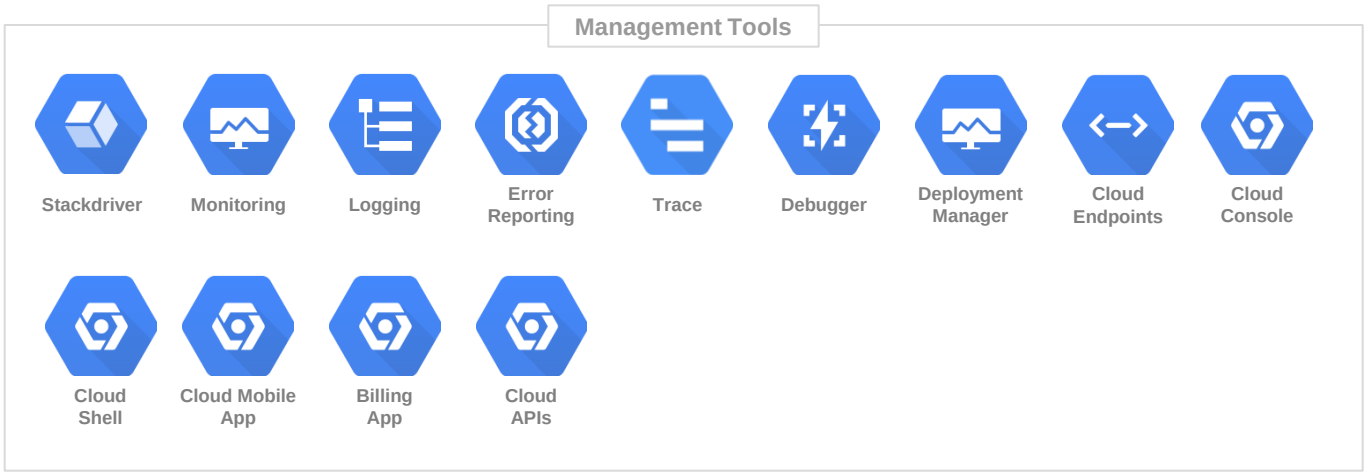
Google App Engine

An example of PaaS

Google App Engine (GAE) – a part of the Google Cloud Platform (GCP)



Other tools in GCP

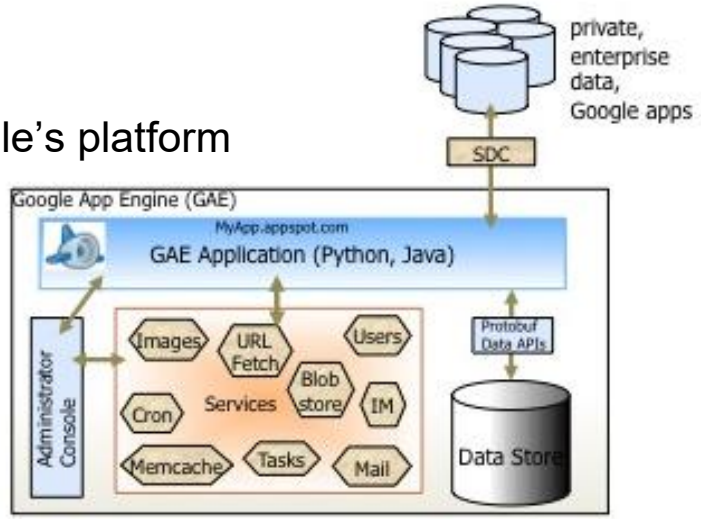


Google App Engine - overview

- Write your application with a set of useful library supports
  - I.e., “Platform as a service” (PaaS)
  - E.g., Memcache, Mail, OAuth/OpenID, Blobstore, etc.

- Supports Java, Python, PHP, Node.js, and Go
  - Many third-party frameworks, e.g., Laravel, WordPress, etc., can run in GAE as well and benefit from its great scalability.

- Test programs locally in your machine, deploy on Google’s platform



  - Automatic scaling – a big advantage
  - => GAE serves billions of page views per day
  - Pay-per-use - free-of-charge for low-volume use

- Example cases - Rovio and Angry Birds:

  “Our web games tend to be popular immediately, so we don’t have the option of scaling them over time. Google App Engine makes the process painless, since it can instantly launch as many servers as we need.”

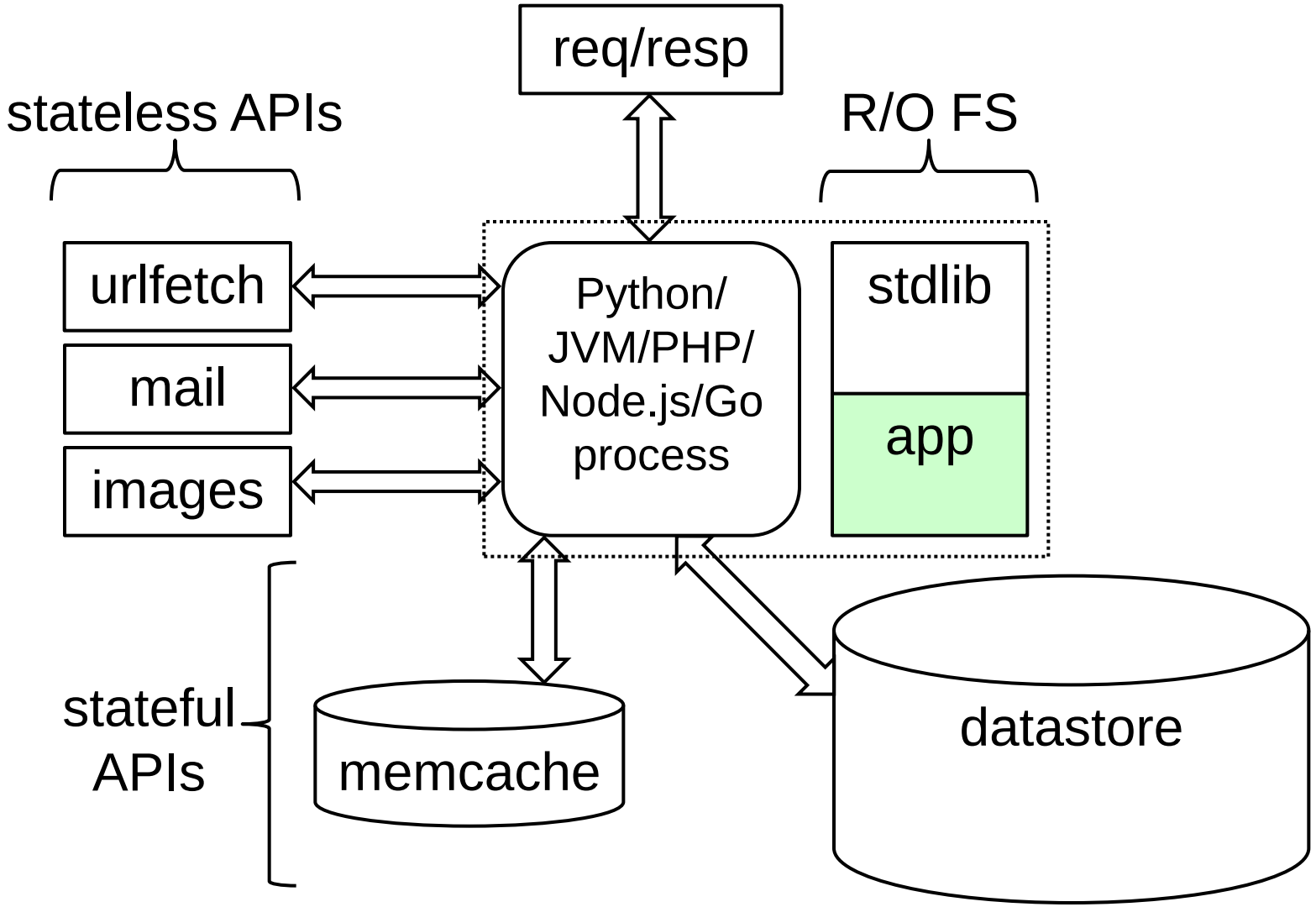
- Snapchat was also built on Google App Engine

GAE library support examples

- OpenID
  - A decentralized single sign on service.
  - For users to have one identity that can be used for authentication in multiple websites
  - Managed by the OpenID Foundation

- Other examples: Translation API, Natural Language API (for speech recognition), etc.

GAE architecture



Web application hosting

- App Engine handles HTTP(S) requests, nothing else
- No performance tuning required
  - Scalability (multithreading) is supported automatically
- Example cost (can set daily quota):

Check online for updated costs and the daily free quotas

Resource	Unit	Unit cost (in US $)
Instances*	Instance hours	$0.05
Outgoing Network Traffic	Gigabytes	$0.12
Incoming Network Traffic	Gigabytes	Free
Datastore Storage	Gigabytes per month	$0.18
Blobstore, Logs, and Task Queue Stored Data	Gigabytes per month	$0.026
Dedicated Memcache	Gigabytes per hour	$0.06
Logs API	Gigabytes	$0.12
SSL Virtual IPs** (VIPs)	Virtual IP per month	$39.00
Sending Email, Shared Memcache, Pagespeed, Cron, APIs (URLFetch, Task Queues, Image, Sockets, Files, and Users)		No Additional Charge

Using Google App Engine

- Download Java 8
- Download the Eclipse IDE for Java EE and install Google Cloud Tools for Eclipse
- Develop your program locally and debug locally
  - **Quite a number of Java MVC frameworks (e.g., Spring MVC, etc.) supported**
- Register a user account at Google
- Obtain a unique application ID ([app-id])
  - Your application will then be available at https://[app-id].appspot.com/
- Upload the code
- Manage the application using the Administration Console
  - View error logs, traffic logs
  - Switch between different versions of your application
- (More on GAE in the demo)

How to choose cloud services?

- Most public clouds offer similar essential features, which fulfil the needs of most websites / web applications
  - AWS's EC2 vs GCP's Compute Engine vs Microsoft Azure Platform, etc.
  - AWS's SimpleDB vs GCP's Cloud Datastore ... ...
- Some cloud providers are stronger in certain areas
  - E.g., AWS has the widest range of cloud tools; Google is strong in PaaS with GAE; IBM Cloud has a set of powerful APIs for AI and machine learning; Google has a Translation API, etc.
- Pricing; and different clouds offer different types of discounts – check to decide
- A multi-cloud (or hybrid cloud) strategy, if that is feasible, can minimize vendor lock-in and simplify service failover or disaster recovery
  - Most websites don't rely on specialized APIs => good candidates for using hybrid clouds
  - Two possibilities:
    - Cloud A + Cloud B; or
    - Your on-premise, private cloud (or ordinary web servers) + a cloud service
  - Design your web application so that it can be run in, or migrated to, another cloud easily
    - **Separate** your implementation into two parts: a platform-dependent part (e.g., DB access routines, runtime library configurations), and a platform-independent part (e.g., your application logic)
  - Tools like OpenStack and AppScale allow you to run workloads in multiple, public and private clouds

The Flexera State of the Cloud Report 2020

- An updated survey/report on the major players in cloud computing, their relative popularities, how users are using them and some latest trends
  - E.g. they even have an analysis on the impact of COVID-19 towards cloud usage
- Good for understanding the field from a management perspective
- Free to download:

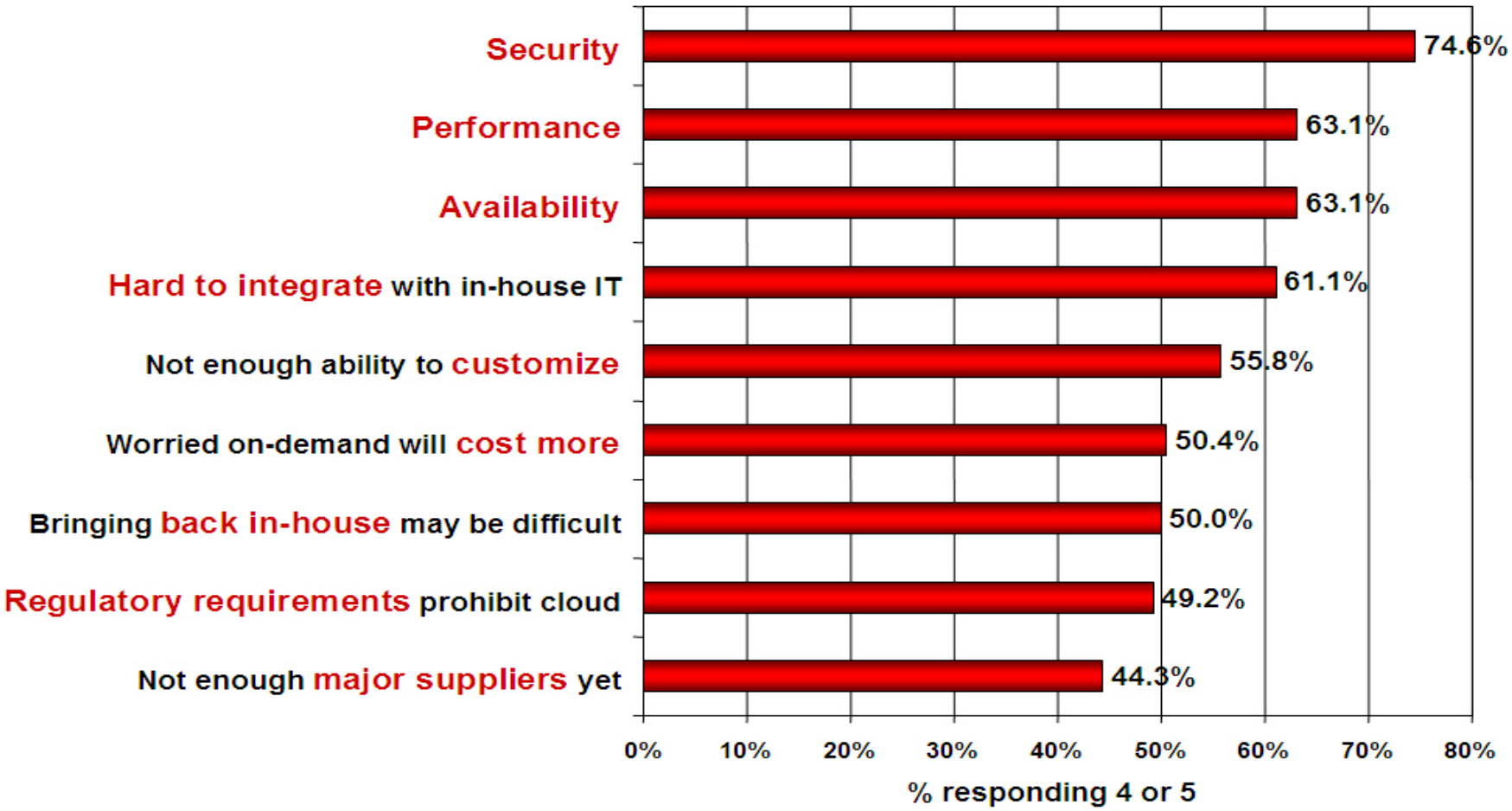
https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020

Security issues of using cloud services

Security the main concern – CTOs' view

Q: Rate the challenges/issues ascribed to the 'cloud'/on-demand model

(1=not significant, 5=very significant)



Source: IDC Enterprise Panel

Advantages (for security)

- Using cloud computing seems to bring security concerns, but in fact, cloud computing can bring improvements to security as well
- Shifting public data to an external cloud reduces the exposure of your internal data
- Cloud's homogeneity makes security auditing/testing simpler
  - E.g., can secure a master image before replicating it to multiple VMs
- Dedicated security team
  - 24x7 supports
  - Computing environments are always kept updated at the desired level, e.g., SaaS, PaaS, etc.
- Data security
  - Automatic data replication and backup
  - Data encryption
- Network security
  - Comprehensive security supports (IDS, firewall, authentication)
  - Distributed denial of service (DDOS) protection
  - VLAN/VPN capabilities

Security concerns

- Possibility for massive outages
- You have to fully understand the vendor's security model
- Can all kinds of access be logged and checked?
  - Some proprietary implementations can't be checked
- Lower down the stack (SaaS => PaaS => IaaS) the cloud vendor provides, the more security issues the clients have to take care of by themselves
- For VMs, application multi-tenancy
  - Could an attacker "escape" from a guest virtual machine instance to the host OS?
- Compatibility of code/library/security updates

Additional concerns

- Data encryption is crucial for cloud computing
  - Always encrypt all a) administrative accesses; b) access to applications; and c) data in storage and backup media
  - Limit access to the secure key stores, plan for key backup and recovery
- Other data security concerns
  - Important data should be hosted on clouds only when you have enough access control, and use data encryption, etc.
  - Also need to check foreign governments' policies and privacy / data ownership laws
- Most cloud providers provide firewalls, but it is (**even more**) important to have your systems (including VMs, OS and applications) well-configured, updated and monitored
- Contingency planning and disaster recovery

Summary of Part 3 of the course – "Interoperability" (Sessions 7-9)

- Web 2.0 promotes data reuse across websites, which rely on **web APIs** using standard protocols and standard data formats
  - We have discussed how to design (and consume others') web APIs and mashups
  - JavaScript-based, Resource-/Service-oriented, RPC-based and Feed-based architectures
  - Web APIs can be implemented with web service toolkits (e.g., Apache Axis, Java EE, etc.), and many other popular MVC frameworks
  - Many web APIs form the "SaaS" element in cloud computing
- Typical usage of web APIs:
  - Creating website components (mashups) – greatly simplify website development by "gluing" data and components together to form the required functions
    - Common motivations of mashups – data combination, visualization and aggregation
  - Sharing data with remote websites by offering a web API
  - Client-side rendering – React as one example
    - Considerations for choosing between server- and client-side rendering frameworks discussed
- Cloud computing introduced, which provides:
  - Infrastructures (IaaS);
  - Development/runtime platforms (PaaS); or
  - Software components (SaaS)

Summary of Part 3 of the course - "Interoperability" (Sessions 7-9)

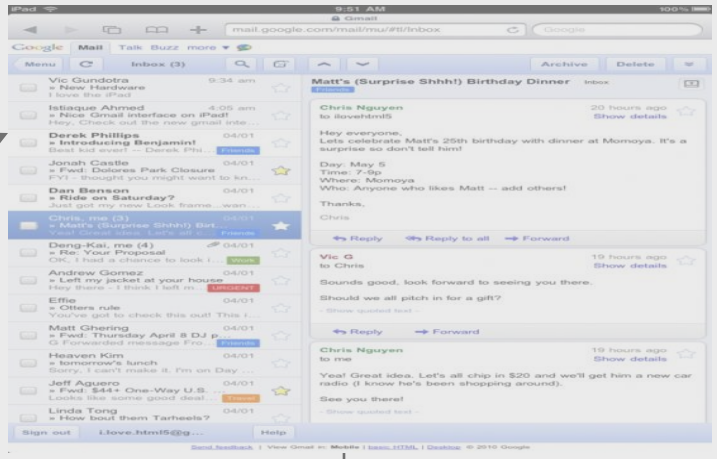
- Benefits of cloud computing:
  - Pay-per-use model => cost-effectiveness. No need to purchase hardware for the "peak demands"
  - Computing facilities are well managed at the desired level; developers can focus on the application logic
  - Simplified website/app development/deployment:
    - Scalability and availability are well taken care of – developers can save a lot of effort.
    - Some PaaS platforms provide handy development tools and library supports (e.g., DB, OpenID, etc.)
    - Some PaaS services can be used together with the server-/client-side frameworks we've discussed in Part 2
- AWS (PaaS+IaaS)/ GAE (PaaS) introduced
- Security is a main concern in cloud computing; some issues discussed

Scope revisited

Part 1 (lectures 1-3):

Websites becoming "web apps" -> more sophisticated, lots of interactions with users, "Web 2.0", etc.

Part 2 (lectures 4-6):



Integration and interoperability issues -> how to reuse existing & remote data?

Part 3 (lectures 7-9):

YouTube, Gmail, Amazon, online databases, Maps, updated event lists, YOUR websites, ……

The web/cloud(s)

Part 4 (lecture10):

You have a great website, how to make it loaded fast at users' computers, and most important… popular?

Optimizations

Post-class self-learning resources

- Post-class readings:
  - W3Schools.com's tutorial on React
  - Take a brief look at the following client-side MV* frameworks:
    - Angular, Vue.js, Ember.js, Backbone.js
    - Also see TodoMVC.com for more examples
  - Take a brief look at some popular clouds for website/webapp development:
    - Amazon Web Services
    - Google Cloud Platform and Google App Engine
    - Microsoft Azure Cloud Computing Platform
  - Take a brief look at OpenStack and AppScale, tools for running workloads in multiple clouds
  - The Flexera State of the Cloud Report 2020

- References:
  - A. Freeman. Pro React 16. Apress. 2019.
  - Introduction to the ATOM Syndication Format by W3C
  - RSS 2.0 and ATOM 1.0 compared

- Please see Moodle for the links.