ITMD 441/541
Web Application Foundations

Week 13

FALL 2023 – NOVEMBER 21, 2023

Agenda

- Discuss some backend dev concepts

Backend Dev Concepts

Front-end vs Back-end

- Every web application can be broken down into the two part
- Front end is what is delivered to the user's browser for the user to interact with
- Back end is what code is running on the server(s) that handles much of the business logic and responds to requests from and interacts with the front end
- Why do we use back-end code?
  - Security – Some of our code needs to be able to be trusted
    - Data validation, data security, user/session management, authorization, etc
  - Performance
    - More resources for data-intensive computation that can happen closer to the data store
    - Ability to avoid processing duplicate requests with caching at different layers
  - Compatibility
    - We can make a universally compatible API for other front-ends or deliver dynamic content by rendering the HTML templates on the server before delivering them to the client.

Front-end vs Back-end

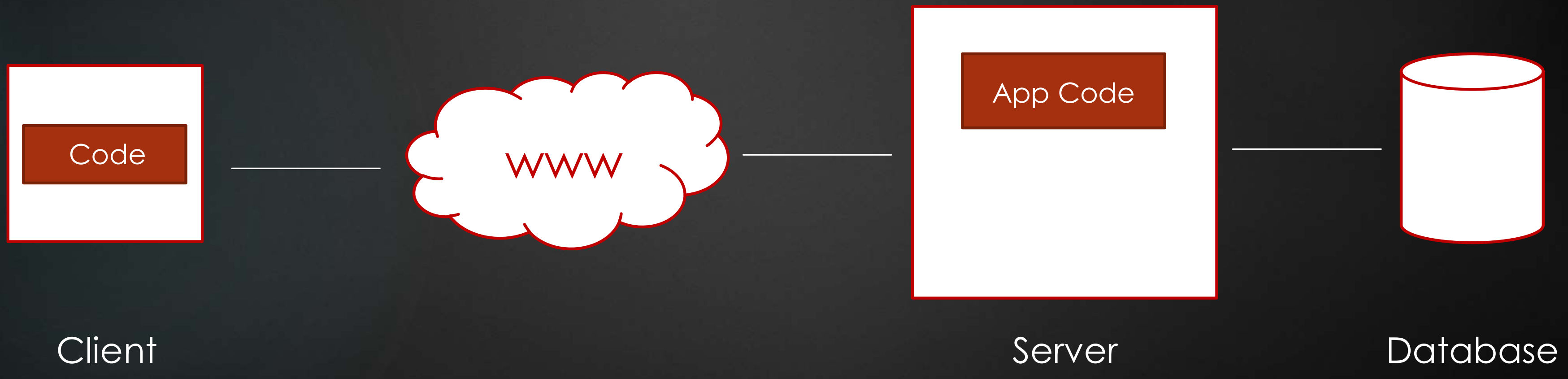
- There is some considerations for where we put our logic and parts of our application
- The front-end typically has the UI and presentation that the user interacts with and some logic
- The back-end typically handles persistent storage to something like a database and all the application business logic
- The back-end would also provide access to some type of API for the front-end whether that is a traditional web API or something like a JSON API.
- Logic in the front-end can be very responsive for the user but can have considerations for security, capabilities, performance, and code reuse.
- Logic in the back-end is much better handling security, heavy computation, compatibility, and can easily be used with different clients or front-ends

Back-end Components

- Servers / Infrastructure
- Databases / Persistent Storage
- Languages
- Frameworks
- Architectures
- APIs

Servers / Infrastructure

- Traditional basic setup consisting of a server running all the app code, a database that the server talks to, and the client
- This would be considered a simple monolithic design

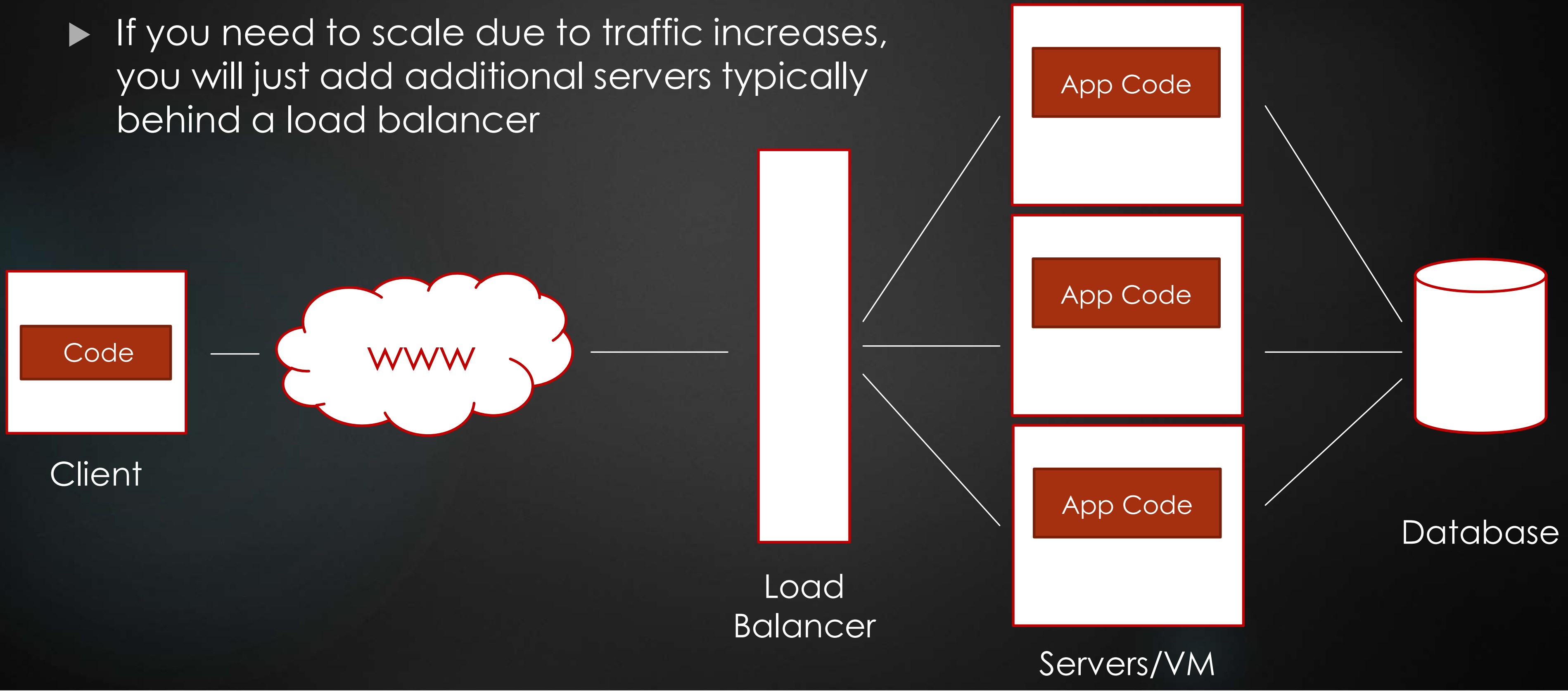


Servers / Infrastructure

- These days most companies rent server capability (VMs) from cloud providers instead of owning and managing their own hardware
- IaaS - Infrastructure as a Service
  - VMs and other virtualized hardware is used instead of physical hardware
  - Additional VMs can be launched if capacity is needed vs buy more hardware
- PaaS – Platform as a Service
  - Cloud providers handle all the underlying VMs and all the company needs to due is upload their backend app code.
  - Google App Engine, Azure App Service, Amazon Elastic Beanstalk
- SaaS – Software as a Service
  - A company provides the backend and an API others can use

Servers / Infrastructure

- If you need to scale due to traffic increases, you will just add additional servers typically behind a load balancer



Servers / Infrastructure

- What problems does this monolithic design bring?
- Each server/vm needs the entire app code base and all dependencies or modules being used
- Updating one part of the application requires us to redeploy the entire application
- What is one part of the application is just needing more resources?
- These issues and more have brought about the popularity of microservices architecture and serverless applications

Microservices

- App gets broken up into smaller apps that focus on one particular part of the application
- Each of these smaller apps have their own infrastructure and it can be the same or different from the others
- Each of these services provide an API to allow them each to communicate with each other
- They can be added, upgraded, reused, and developed independently
- There are companies out there that provide these small services that you can plug into your application. For example, there is twilio that runs an email sending and messaging service.

Serverless

- This is a cloud computing model where the app does not run on dedicated VMs or containers that the customer manages
- Does not continuously use resources
- Resources are used only during the processing of a request
- Typically, code is deployed as a cloud function that runs on certain requests
- The cloud vendor provides the underlying infrastructure and will scale up as needed.

Persistent Data Storage

- Primary Databases
  - MySQL
  - PostgreSQL
  - Oracle
  - Microsoft SQL
  - MongoDB
- Additional Data Storage Technologies
  - Caching with redis, memcache, varnish, or others
  - Cloud Database services
  - Cloud File BLOB storage

Popular Languages

- Early backend web languages like PHP and ASP allowed you to mix code logic in your HTML files to provide dynamic content which led to what some call spaghetti code
- Newer versions of those language and the others tend to have a more structured approach
- Most languages now have package managers that are used to manage dependencies

- PHP - Composer
- ASP / C# - NuGet
- JavaScript (NodeJS) - npm
- Python - Pip
- Ruby – RubyGems, Bundler
- Java / JSP - Maven

Popular Frameworks

- Now days most people start with a web framework.
- More secure and quicker than writing all the pieces from the ground up

- PHP – Laravel, Symfony, CakePHP, CodeIgniter, Zend Framework, and more
- ASP / C# - ASP.net and MVC
- JavaScript (NodeJS) – Express, NextJS
- Python – Django, Flask, web2py, Pylons, Pyramid, and more
- Ruby – Ruby on Rails
- Java / JSP – Spring, JSF, and more

APIs (start here for day 2)

- API – Application Programming Interface
- The public interface for interacting with a back-end
- The list of all the different types of requests and their endpoints
- Requests that are not allowed should respond with an error
- Various types of API design that use different methods and URIs
  - **REST representational state transfer – most common, used for exchanging data and resources**
  - SOAP - simple object access protocol – messaging standard by W3C and uses XML
  - RPC – remote procedure call – used to invoke actions or processes and uses JSON or XML
  - GraphQL – Uses POST at a single endpoint for all queries

REST

- Type of request has special meaning and url path can be the same or different from other types
- APIs that use this naming convention and follow the other aspects of REST are considered a RESTful API or REST API
- Defined by Roy Fielding in his 2000 Ph.D. dissertation
  - Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- Fielding helped design HTTP 1.1 and generalized URLs to URIs using this research

REST

- **Six architectural constraints**
  - Uniform interface
    - A resource in the system should have only one logical URI, and that should provide a way to fetch related or additional data
  - Client-server
    - Client and server applications must be able to developed independently
  - Stateless
    - each request contains all information necessary to service request and the response
  - Cacheable – High performance
  - Layered system
    - client cannot determine if it is connected to end server or intermediary along the way
  - Code on demand (optional)
    - Most of the time you would send a resource representation using XML or JSON but you can return executable code

REST → CRUD

- POST = create something - Create
  - POST /contacts
- GET = get something - Read
  - GET /contacts
  - Safe method with no side effects only retrieves data
- DELETE = delete something - Delete
  - DELETE /contacts/2
- PUT = update/replace something - Update/Replace
  - PUT /contacts/2
- PATCH = update/modify something – Update/Modify
  - PATCH /contacts/2

Main Actions	HTTP methods
Create→	POST
Read →	GET
Update →	PUT
Delete →	DELETE

REST Example

You work at an ice cream shop and you're trying to build a web application to show the flavors of ice cream that are in stock that day and allow the workers to actually make updates to those flavors

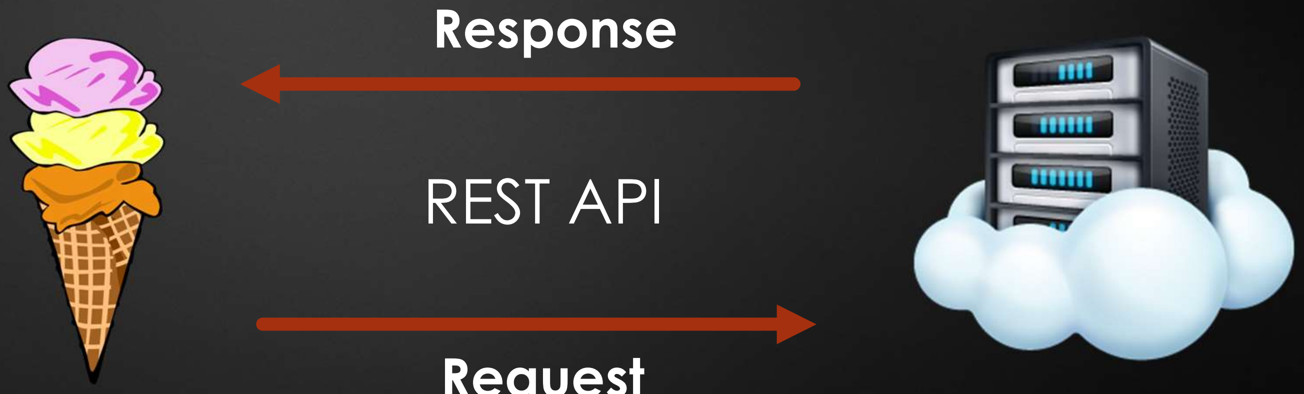
How do you do this?

- With a REST API
- You have your web app or web page communicate with a cloud-based server via a REST API

REST Example

Example of an endpoint

HTTP://icecream.com/api/**flavors** **(note flavors is a resource)**



REST Example

HTTP://icecream.com/api/**flavors** **(note flavors is a resource)**

REQUEST

Header	
Operation	**End Point**
Parameter/Body	

RESPONSE

REST Example

HTTP://icecream.com/api/**flavors** (note flavors is the resource)

Step 1. We want to retrieve the flavors. What does our REST API request look like?

REQUEST

Header *This might be an API Key or some authentication data*	
Operation GET	End Point /api/flavors
Parameter/Body	

RESPONSE

```
[ { "id" : 0,
"flavor":  "strawberry"},
{ "id" : 1,
"flavor":  "mint choc"}  ]
```

REST Example

HTTP://icecream.com/API/**flavors** **(note flavors is the resource)**

Step 2. What happens when mint chocolate runs out? We want to replace it with chocolate.

REQUEST

Header	
Operation PUT	**End Point** /api/flavors/1
Parameter/Body {"flavor": "chocolate"}	

ID is 1

RESPONSE

{ "id" : 1,
"flavor": "chocolate"}

REST Example

HTTP://icecream.com/API/**flavors** (note flavors is the resource)

Step 3. We just received a new flavor called Lemon Raspberry. We want to load this new ice cream up into your website. So, we are creating.

REQUEST

Header	
Operation POST	End Point /api/flavors
Parameter/Body {"flavor": "Lemon Raspberry"}	

No ID

RESPONSE

{ "id" : 2,
"flavor": "Lemon Raspberry"}

Assignments

Reading/Assignments

- Week 13 and Week 14 questions will be included in the final exam.
- There will be no additional lab
- Final Exam will be released before Dec 4 and will be due by end of day December 7
- Same format as Midterm
- **Two hours to complete 55 questions**