FULL STACK ENGINEERING

Birla Institute of Technology & Science, Pilani

Work Integrated Learning Programmes Division

Second Semester 2023-2024 Mid-Semester Test

(EC-2 Regular)

Course No.	: SE ZG503
Course Title	: Full Stack Application Development
Nature of Exam	: Closed Book
Weightage	: 30%
Duration	: 2 Hours
Date of Exam	: 17/03/2024 (FN)

No. of Pages = 3

Note to Students:

1. Please follow all the *Instructions to Candidates* given on the cover page of the answer book.
2. All parts of a question should be answered consecutively. Each answer should start from a fresh page.
3. Assumptions made if any, should be stated clearly at the beginning of your answer.

Q.1 Design a set of RESTful APIs for a mobile app that helps users find nearby restaurants for food delivery or dine-in. The app should provide autocomplete suggestions as users type in their search queries, allow them to search for restaurants based on various criteria, view detailed information about individual restaurants, and read reviews before making a decision.

Below are the detailed requirements

1. Autocomplete Search:

- Users should receive autocomplete suggestions as they type in the search box.
- Autocomplete suggestions should be based on the partial search term entered by the user.

2. Restaurant Search:

- Users should be able to search for restaurants based on keywords.
- The search should take into account the user's current location.
- Users should have the option to filter restaurants based on delivery availability.

3. Restaurant Details:

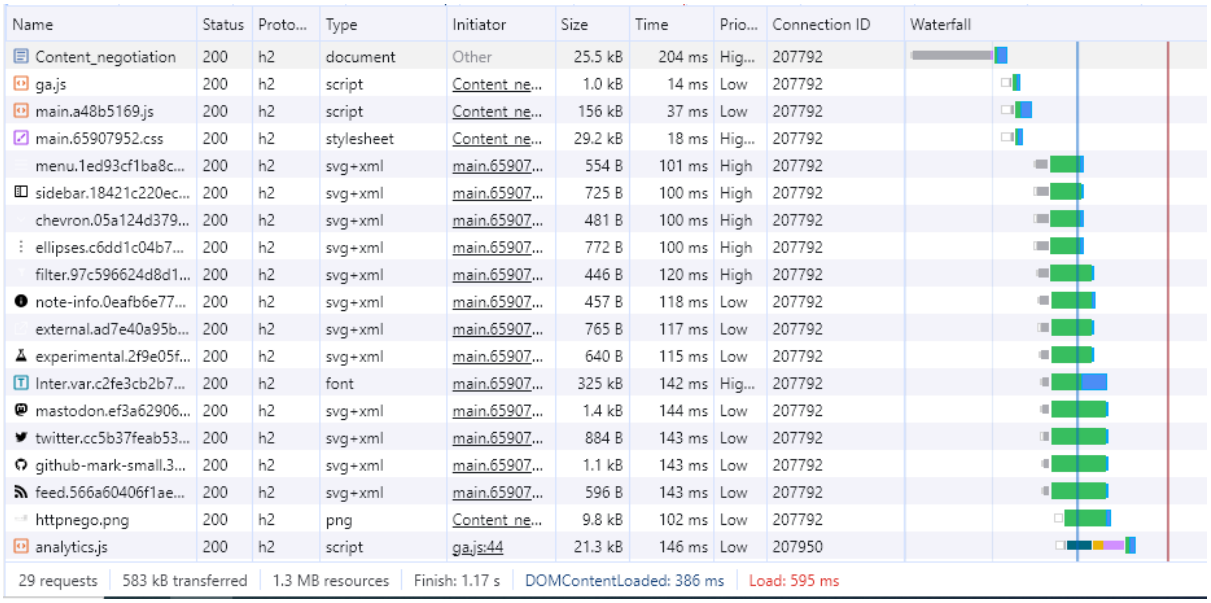
- Users should be able to view detailed information about a specific restaurant.
- Information should include the restaurant's name, address, delivery availability, and rating.

Design the following API endpoints: [3*3=9 Marks]

I. Autocomplete Search
II. Restaurant Search
III. Restaurant Details

For each API endpoint, Specify the purpose and the exact URL, Identify the supported HTTP operation, define required and optional parameters with examples, and describe the structure of the response along with the appropriate status code with an example output.

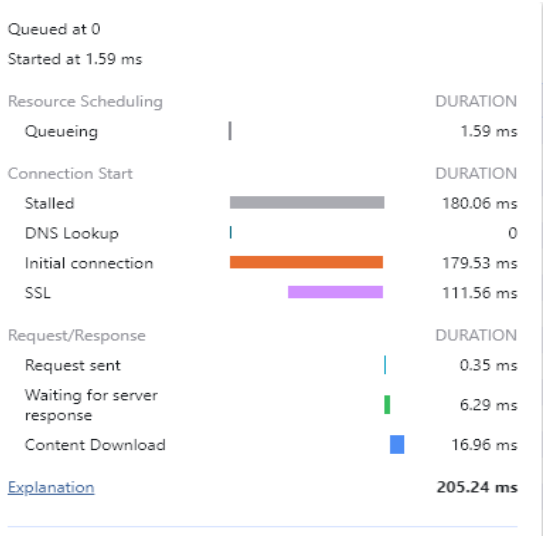
Q.2 The image below is a waterfall view from the Chrome developer tools of a webpage. Analyze the snapshot given and answer the following questions **[5 Marks]**



Name	Status	Proto...	Type	Initiator	Size	Time	Prio...	Connection ID	Waterfall
Content_negotiation	200	h2	document	Other	25.5 kB	204 ms	Hig...	207792	
ga.js	200	h2	script	Content_ne...	1.0 kB	14 ms	Low	207792	
main.a48b5169.js	200	h2	script	Content_ne...	156 kB	37 ms	Low	207792	
main.65907952.css	200	h2	stylesheet	Content_ne...	29.2 kB	18 ms	Hig...	207792	
menu.1ed93cf1ba8c...	200	h2	svg+xml	main.65907...	554 B	101 ms	High	207792	
sidebar.18421c220ec...	200	h2	svg+xml	main.65907...	725 B	100 ms	High	207792	
chevron.05a124d379...	200	h2	svg+xml	main.65907...	481 B	100 ms	High	207792	
ellipses.c6dd1c04b7...	200	h2	svg+xml	main.65907...	772 B	100 ms	High	207792	
filter.97c596624d8d1...	200	h2	svg+xml	main.65907...	446 B	120 ms	High	207792	
note-info.0eafb6e77...	200	h2	svg+xml	main.65907...	457 B	118 ms	Low	207792	
external.ad7e40a95b...	200	h2	svg+xml	main.65907...	765 B	117 ms	Low	207792	
experimental.2f9e05f...	200	h2	svg+xml	main.65907...	640 B	115 ms	Low	207792	
Inter.var.c2fe3cb2b7...	200	h2	font	main.65907...	325 kB	142 ms	Hig...	207792	
mastodon.ef3a62906...	200	h2	svg+xml	main.65907...	1.4 kB	144 ms	Low	207792	
twitter.cc5b37feab53...	200	h2	svg+xml	main.65907...	884 B	143 ms	Low	207792	
github-mark-small.3...	200	h2	svg+xml	main.65907...	1.1 kB	143 ms	Low	207792	
feed.566a60406f1ae...	200	h2	svg+xml	main.65907...	596 B	143 ms	Low	207792	
httpnego.png	200	h2	png	Content_ne...	9.8 kB	102 ms	Low	207792	
analytics.js	200	h2	script	ga.js:44	21.3 kB	146 ms	Low	207950	

29 requests | 583 kB transferred | 1.3 MB resources | Finish: 1.17 s | DOMContentLoaded: 386 ms | Load: 595 ms

The below image shows the time taken for the first request (content negotiation)



i) Which version of HTTP does the website use, and what is the number of request-response cycles?

ii) Does the webpage reuse the TCP connection for multiple requests? Justify your answer.

iii) What is the significance of TTFB(Time to First byte), and what is the value of TTFB for the first request?

iv) What does it mean if long blue sections(Content download) are present in the waterfall view of the request, and how can it be reduced?

v) How much time does the webpage take to complete HTML Parsing?

Q.3 A frontend react app polls the backend REST API every five minutes to check for changes in the user database. Instead, I want the backend to detect a change in the database and automatically send that updated information to the React app.

Can I use Webhooks to implement this scenario? Suggest a solution to implement this scenario. [3 Marks]

Q.4 HTTP/1.x sometimes results in a slow page load and poor user experience. Does the HTTP/2 resolve this problem, and how? [4 Marks]

Q.5 What challenges are associated with cross-platform app development, and how can they be mitigated? [3 Marks]

Q.6 [3*2=6 Marks]

I. How does the "ETag" header work in HTTP caching, and what role does it play in cache validation?

II. What response/s should the REST API send in the case of a failed content negotiation, where the client and server cannot agree on a mutually acceptable representation format?

III. In what scenarios will the API send a 204 No content status code?

Birla Institute of Technology & Science, Pilani

Work Integrated Learning Programmes Division

Second Semester 2023-2024 Comprehensive Examination

(EC-3 Regular)

Course No.	: SE ZG503
Course Title	: Full Stack Application Development
Nature of Exam	: Open Book
Weightage	: 40%
Duration	: 2 ½ Hours
Date of Exam	: 19/05/2024 (FN)

No. of Pages = 3

Note to Students:

4. Please follow all the *Instructions to Candidates* given on the cover page of the answer book.
5. All parts of a question should be answered consecutively. Each answer should start from a fresh page.
6. Assumptions made if any, should be stated clearly at the beginning of your answer.

Question 1:

Answer the below questions related to gRPC. [3+4=7]

a) Explain how bidirectional streaming works in gRPC, and identify the scenarios that can benefit from it?
b) Write a protobuf file to represent the following Todo service. The Todo service has the three methods. These methods allow clients to create, read, and stream todo items.
   - createTodo: Accepts a TodoItem and returns a TodoItem.
   - `readTodos`: This method returns a list of Todoitems
   - readTodosStream: This method returns a stream of todo items

Question 2:

Answer the below questions related to Security. [4+4=8]

a) You're developing a social media management platform that allows users to schedule posts and analyze engagement metrics across multiple social media platforms, including Facebook, Twitter, and Instagram. Explain the flow of acquiring access to user data from these platforms through OAuth, and recommend the most suitable grant type for your application.

b) A user wants the smart TV to play his/her favorite morning playlist from Spotify. Explain the flow of acquiring access to the Spotify account through OAuth, and recommend the most suitable grant type for this scenario. Explain if API Keys are a better option that OAuth in this case

Question 3

How do you test and ensure that APIs adhere to their specified contract, such as RESTful API standards or OpenAPI specifications? [3

Question 4:

Identify the react antipattern/s in the code below and rewrite the code using best practices. Describe briefly what kind of problems will the antipatterns could create. [5]

```
import { useState } from 'react';
const ComponentA = () => {
    const [count, setCount] = useState(0);
    const items =['apple', 'banana', 'orange'];

    const ChildComponent = () => (
        <ul>
        {items.map((item, index) => <li>{item}</li>)}
        </ul>
    );
    return (
     <div>
        <ChildComponent />
        <button onClick={() => setCount(count + 1)}>Count me</button>

     </div>
    );
};
export default ComponentA;
```

Question 5

Describe what extended validation certificates have in addition to regular certificates and what problem EV certificates are trying to address. [3]

Question 6

"Single page application model breaks the browser's design for page history navigation using the Forward/Back buttons." Explain the reason and list the workaround options available in frontend frameworks like ReactJ for the same [4]

Question 7

Node.js runs JavaScript code in a single-threaded event loop, which means that all JavaScript code is executed sequentially in a single thread. Explain how it can handle multiple operations concurrently without creating additional threads. [5]

Question 8

Below is the HTML code of a simple website with several accessibility issues. Analyze the code and identify the accessibility issues. [Note: Assume that all the relevant CSS for the below code are present]
[5]

```
<html>
<body>
    <div id="header">
        <div id="nav">
            <a href="https://example.com"><img id="logo" src="logo-oneline.svg" /></a>
            <div id="breadcrumbs">
                <ul>
                    <li><a href="https://example.com">Division of Information Technology</a></li>
                    <li><a href="https://example.com">Web Development</a></li>
                    <li><a href="https://example.com">Making Websites</a></li>
                </ul>
            </div>
        </div>
    </div>
    <div id="main">
        <div id="content">
            <div class="h1">Example Inaccessible Web Page</div>
            <p>This is an example of an <u>inaccessible</u> web page.</p>
            <div class="h2">HTML Head Section</div>
            <ul>
                <li class="alert">The <code>!DOCTYPE html</code> tag specify a document type. </li>
                <li>The <code>title</code> tag specifies a unique page title</li>
                <li>The <code>link</code> and <code>script</code> tags include styles and scripts.</li>
            </ul>
        </div>
        <div id="aside">
            <div class="h2">View Source Code</div>
            <ul>
                <li><a target="_blank" href="www.example.com">View the HTML source of this page</a></li>
                <li><a href="https://developer.chrome.com/devtools">Chrome Developer Tools</a></li>
            </ul>
        </div>
        <div id="footer">
            <div>
                <ul>
                    <li><a href="http://disclaimer">Disclaimer</a></li>
                    <li><a href="http://privacy">Privacy Policy</a></li>
                </ul>
            </div>
        </div>
</body>
</html>
```

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

WORK INTEGRATED LEARNING PROGRAMMES

Course Handouts

Part A: Content Design

Course Title	**Full Stack Application Development**
Course No(s)	SE ZG503
Credit Units	4
Content Authors	Akshaya Ganesan
Version	1.0
Date	September 2023

Course Description:

The modern software application landscape is evolving rapidly, moving from the conventional, layered web applications hosted on remote servers to the mobile-only / mobile-first and cloud-native applications with complex deployment options. At the same time, the software development teams have started adapting the most agile development methodologies, enabling them to deliver the software with better quality at shorter and more frequent intervals. These developments have resulted in the necessity of engineers having multiple skill sets essential for every aspect of the software development life cycle, right from requirement analysis to application deployment.

This course aims to provide a comprehensive introduction to modern Web application architecture approaches, frontend and backend technologies, and suitable web application frameworks required for developing modern web apps. It focuses on designing and implementing end-to-end functional web applications, learning the key patterns followed at each layer of the application architecture and technology considerations to choose an appropriate implementation technology. The course also involves hands-on exposure to full-stack development of web applications using development frameworks.

Course Objectives

No	Course Objective
CO1	Understand the modern application landscape and the evolution of the application landscape.
CO2	Build an understanding of an end-to-end application's typical structure, design, and implementation considerations.
CO3	Comprehend the necessity and usefulness of the client and server-side frameworks, along with their strengths and weaknesses.

CO4	Develop and test a working model of web application using the tech stack.

Text Book(s)

T1	Web Development with Node and Express by Ethan Brown , Oreilly Media 2nd Edition , 2019

Reference Book(s) & other resources

R1	Building Microservices: Designing Fine-Grained Systems Book by Sam Newman, 1st edition, published by O'Reilly Media, Feb 2015
R2	The Design of Web APIs by Arnaud Lauret Published by Manning Publications; 1st edition (November 2019)
R3	Full Stack Web Development: The Comprehensive Guide by Philip Ackermann Shroff/Rheinwerk Computing; First Edition (2 August 2023)
R4	Microservice APIs: Using Python, Flask, FastAPI, OpenAPI and More. Peralta, J. H. (2023): Manning.
R5	GRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes Book by Danesh Kuruppu and Kasun Indrasir, Oreilly Media
Web Resources	Mozilla Developer Network https://developer.mozilla.org/en-US/ React JS https://react.dev/learn

Content Structure

Module 1: Application Development

- Introduction to the various Application Landscape: [Web applications, Mobile applications, Cross Platform applications, Cloud native applications, Serverless Applications]
- Layered Architectures (client /server, 2/3 tier- N tier)
- Monolithic
- Distributed Architectures – Service-oriented architecture, Microservices
- MVC Pattern

Module 2: Understanding the Basics

- Structure of web applications
  - Frontend (HTML, CSS, JavaScript),
  - Backend Server Side logic, API, Web Services
  - Database
- Client – Server Communication
- Relationship between URLs, Domains, and IP Addresses
- Domain Name Systems, Content Delivery Networks
- Technologies and Tools for Full stack development

Module 3: Web Protocols

- HTTP

    - HTTP Request- Response and its structure
    - HTTP Methods;
    - HTTP Headers
    - Connection management - HTTP/1.1 and HTTP/2
- Synchronous and asynchronous communication
- Communication with Backend
    - AJAX, Fetch API
    - Webhooks
    - Server-Sent Events
- Polling
- Bidirectional communication - Web sockets

Module 4: Server Side: Implementing Web Services

- REST
    - Principles of REST
    - REST constraints
    - Service Design with REST
        - Interaction Design with HTTP
        - Interface Design (URI)
        - Representation and Metadata design
    - Implementing REST API
        - Using a Framework
        - URL Mapping
        - Routing Requests-Redirection
        - Implementing a web server
        - Processing request, response, data
    - Storing data in databases
        - Models
        - Object Relational Mapper
        - Interaction with DBs
    - API versioning and documentation

- GraphQL
    - Schemas and Types
    - Thinking in Graphs
    - Serving over HTTP
    - Implementing the GraphQL API
    - Validation and Execution
- gRPC
    - Service Definition- Protobuf
    - Architecture
    - Channels Streaming and Types

Module 5: Securing Application

- Basic Authentication
- API Authorization
- JSON Web Tokens

- OAuth
- SSL, TLS and HTTPS
- Common Vulnerabilities

Module 6: Understanding Frontend Development

- Designing and Structuring Webpages
- Making pages Interactive with JavaScript
- Using Browser based Web APIs- DOM, Web storage
- Client-side JavaScript Frameworks: Features and Advantage, MEAN, MERN
- Implementing Single Page Applications using JavaScript Tech stack
    - The Node Ecosystem
    - Project Setup
    - Creating and styling components
    - Managing Component hierarchies and lifecycle
    - Managing State
    - Routing
- Building, deploying and Hosting

Module 7: Testing

- API Testing
    - API Testing and Types
    - Unit Testing
    - Contract Testing
- Frontend
    - Unit testing
    - Cross browser testing
    - Acceptance Testing

Module 8: Accessibility and Performance

- Accessibility
    - Inclusive Design
    - Assistive Technologies
    - Web content accessibility Guidelines
    - Optimizing Websites for Accessibility
    - Testing Accessibility
- Performance
    - Tools and Metrics for Measuring Performance
    - Options for Optimization
        - Caching- Client side, Server side
        - Minifying Code, Compressing files
        - Lazy Loading

Module 9: Latest Advancements

- Progressive Web Apps
- Web Assembly

- Microfronends

Learning Outcomes:

No	Learning Outcomes
LO1	Understand the underlying architecture used for Web applications and identify the various components of the Web Application
LO2	Demonstrate the creation of API to accomplish various backend functionalities of an application like database interaction, handling user requests
LO3	Design and develop user-friendly and interactive web frontends.
LO4	Implement a functional end-to-end web application using client-side and server-side web technologies.

Part B: Learning Plan

Academic Term	First Semester 2024-2025
Course Title	**Full Stack Application Development**
Course No	SE ZG503
Lead Instructor	AKSHAYA GANESAN

Session No.	Topic Title	Study / HW Resource Reference
1	**Module 1: Application Development** • Introduction to the various Application Landscape: [Web applications, Mobile applications, Cross Platform applications, Cloud native applications, Serverless Applications] • Layered Architectures (client /server, 2/3 tier- N tier) • Monolithic • Distributed Architectures – Service-oriented architecture, Microservices • MVC Pattern	R3- Chapter 12, R1 Web references
2	**Module 2: Understanding the Basics** • Structure of web applications ▪ Frontend (HTML, CSS, JavaScript), ▪ Backend Server Side logic, API, Web Services ▪ Database • Client – Server Communication • Relationship between URLs, Domains, and IP Addresses • Domain Name Systems, Content Delivery Networks	R3- Chapter 1, https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/How_the_Web_works

	• Technologies and Tools for Full stack development	
3	**Module 3: Web Protocols** • HTTP ▪ HTTP Request- Response and its structure ▪ HTTP Methods- GET, PUT, POST, DELETE ▪ HTTP Headers ▪ Connection management - HTTP/1.1 and HTTP/2	T1- Chapter 6 R3- Chapter 5 https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
4	**Module 3: Web Protocols(continued)** • Synchronous and asynchronous communication • Communication with Backend ▪ AJAX, Fetch API ▪ Webhooks ▪ Server-Sent Events • Polling • Bidirectional communication - Web sockets	T1- Chapter 8
5	**Module 4: Server Side: Implementing Web Services** • REST ▪ Principles of REST ▪ REST constraints ▪ Service Design with REST ○ Interaction Design with HTTP ○ Interface Design (URI) ○ Representation and Metadata design	R4- Chapter 4 R2
6	**Module 4: Server Side: Implementing Web Services** • REST (continued) ▪ Implementing REST API(NodeJS/Python) ○ Using a Framework ○ URL Mapping ○ Routing Requests-Redirection ○ Implementing a web server ○ Processing request, response, data ▪ Storing data in databases(MongoDB/Postgres) ○ Models ○ Object Relational Mapper ○ Interaction with DBs	R4- Chapter 6 T1- Chapter 13,14
7	**Module 4: Server Side: Implementing Web Services** • REST ▪ API versioning and documentation ▪ Open API ▪ Using swagger • GraphQL	R2, R4- Chapter 5, 8

	▪ Schemas and Types ▪ Thinking in Graphs ▪ Serving over HTTP ▪ Implementing the GraphQL API ▪ Validation and Execution	
8	**Module 4: Server Side: Implementing Web Services** • gRPC ▪ Service Definition- Protobuf ▪ Architecture ▪ Channels Streaming and Types	R5-Chapter 1, 2,3
9	**Module 5: Securing Application** • Basic Authentication • API Authorization • JSON Web Tokens • OAuth • SSL, TLS and HTTPS • Common Vulnerabilities	T1- Chapter 18 R4- Chapter 11
10	**Module 6: Understanding Frontend Development** • Designing and Structuring Webpages • Making pages Interactive with JavaScript • Using Browser based Web APIs- DOM, Web storage • Client-side JavaScript Frameworks: Features and Advantage, MEAN, MERN	https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks
11	**Module 6: Understanding Frontend Development(continued)** • Implementing Single Page Applications using JavaScript Tech stack(React/Angular) ▪ The Node Ecosystem ▪ Project Setup ▪ Package managers- NPM, Module bundlers webpack, Build tools ▪ Creating and styling components ▪ Managing Component hierarchies and lifecycle	https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started
12	**Module 6: Understanding Frontend Development(continued)** ▪ Managing State ▪ Routing ▪ Building, deploying and Hosting	https://react.dev/learn
13	**Module 7: Testing** • API Testing ▪ API Testing and Types ▪ Unit Testing ▪ Contract Testing	R4- Chapter 12

	• Frontend ▪ Unit testing ▪ Cross browser testing ▪ Acceptance Testing	
14	**Module 8: Accessibility and Performance** • Accessibility ▪ Inclusive Design ▪ Assistive Technologies ▪ Web content accessibility Guidelines ▪ Optimizing Websites for Accessibility ▪ Testing Accessibility • Performance ▪ Tools and Metrics for Measuring Performance ▪ Options for Optimization o Caching- Client side, Server side o Minifying Code, Compressing files o Lazy Loading	R3-Chapter 8, 21
15	**Module 9: Latest Advancements** • Progressive Web Apps • Web Assembly • Microfronends	Classroom discussions
16	Recap	

Evaluation Scheme:
Legend: EC = Evaluation Component; AN = After Noon Session; FN = Fore Noon Session

No	Name	Type	Duration	Weight	Day, Date, Session, Time
EC-1	Quizzes	Online		10%	September 1-10, 2024
EC-1	Assignments	Take Home		20%	October 10-20, 2024
EC-2	Mid-Semester Test	Closed Book	2 Hours	30%	Sunday, 22/09/2024 (FN)
EC-3	Comprehensive Exam	Open Book	2.5 Hrs	40%	Sunday, 01/12/2024 (FN)

Note:
Syllabus for Mid-Semester Test (Closed Book): Topics in Session Nos. 1 to 8
Syllabus for Comprehensive Exam (Open Book): All topics (Session Nos. 1 to 16)

Important links and information:

Elearn portal: https://elearn.bits-pilani.ac.in

Students are expected to visit the Elearn portal on a regular basis and stay up to date with the latest announcements and

deadlines.

Contact sessions: Students should attend the online lectures as per the schedule provided on the Elearn portal.

Evaluation Guidelines:

1. EC-1 consists of either two Assignments or three Quizzes. Students will attempt them through the course pages on the Elearn portal. Announcements will be made on the portal, in a timely manner.
2. For Closed Book tests: No books or reference material of any kind will be permitted.
3. For Open Book exams: Use of books and any printed / written reference material (filed or bound) is permitted. However, loose sheets of paper will not be allowed. Use of calculators is permitted in all exams. Laptops/Mobiles of any kind are not allowed. Exchange of any material is not allowed.
4. If a student is unable to appear for the Regular Test/Exam due to genuine exigencies, the student should follow the procedure to apply for the Make-Up Test/Exam which will be made available on the Elearn portal. The Make-Up Test/Exam will be conducted only at selected exam centres on the dates to be announced later.

It shall be the responsibility of the individual student to be regular in maintaining the self-study schedule as given in the course handout, attend the online lectures, and take all the prescribed evaluation components such as Assignment/Quiz, Mid-Semester Test and Comprehensive Exam according to the evaluation scheme provided in the handout.

Evaluation Guidelines:

1. EC-1 consists of two Quizzes. Students will attempt them through the course pages on the Elearn portal. Announcements will be made on the portal, in a timely manner.
2. EC-2 consists of either one or two Assignments. Students will attempt them through the course pages on the Elearn portal. Announcements will be made on the portal, in a timely manner.
3. For Closed Book tests: No books or reference material of any kind will be permitted.
4. For Open Book exams: Use of books and any printed / written reference material (filed or bound) is permitted. However, loose sheets of paper will not be allowed. Use of calculators is permitted in all exams. Laptops/Mobiles of any kind are not allowed. Exchange of any material is not allowed.
5. If a student is unable to appear for the Regular Test/Exam due to genuine exigencies, the student should follow the procedure to apply for the Make-Up Test/Exam which will be made available on the Elearn portal. The Make-Up Test/Exam will be conducted only at selected exam centres on the dates to be announced later.

It shall be the responsibility of the individual student to be regular in maintaining the self-study schedule as given in the course hand-out, attend the online lectures, and take all the prescribed evaluation components such as Assignment/Quiz, Mid-Semester Test and Comprehensive Exam according to the evaluation scheme provided in the hand-out.

Contents

1. Application Introduction

Topic 1.1: Application Development- Introduction

Topic 1.2: Introduction

- This course aims to provide a comprehensive introduction to modern Web application architecture approaches, frontend and backend technologies, and suitable web application frameworks required for developing modern web apps.

Learning Outcomes
Understand the underlying architecture used for Web applications and identify the various components of the Web Application
Demonstrate the creation of API to accomplish various backend functionalities of an application like database interaction, handling user requests
Design and develop user-friendly and interactive web frontends.
Implement a functional end-to-end web application using client-side and server-side web technologies.

Modular Structure

Module 1: Application Development

Introduction to the Various Application Landscape

Module 2: Understanding the Web Basics

Structure of web applications, Client – Server Communication

Module 3: Web Protocols

HTTP, HTTPS, WebSockets

Module 4: Server Side: Implementing Web Services

REST, gRPC, GraphQL

Module 5: Securing Application

JWT, OAuth

Module 6: Understanding Frontend Development

Implementing Single Page Applications using JavaScript Tech stack

Module 7: Testing

API Testing, Unit Testing

Module 8: Accessibility and Performance

Inclusive Design , Assistive Technologies, Tools and Metrics for Measuring Performance

Module 9: Latest Advancements

Progressive Web Apps

Web Assembly

Development Environment

- Tools/Technologies
    - Visual Studio Code
    - Browser(Chrome)
    - Frontend: ReactJS, Node Ecosystem
    - Backend: NodeJS, Express, Django
    - Database: MongoDB/Postgresql
    - Postman for Testing
    - GitHub and GIT

Topic 1.3: Module 1.1: Application Introduction

Topic 1.4: Activity 1

- Open Makemytrip in a browser
- Open the BITS Website in a browser
- Observe the difference in the address bar!!
- Repeat in a different browser
- Install the app from one browser and observe

Topic 1.5: Website

- A group of interlinked web pages having a single domain name
    - Hosted on a web server
    - Accessible over the web with an internet connection
    - Easily accessible through browsers
    - Can be developed and maintained by individuals/teams for personal or business usage
- Use Cases for Static Websites
    - portfolios
    - personal blogs
    - informational websites and
    - small business websites with minimal content updates.
- They are also suitable for landing pages or temporary promotions where the

content doesn't change frequently.

Topic 1.6: Web Application

- A web application is an application program stored on a remote server and delivered over the internet.
- Users can access a web application through a web browser, such as Google Chrome, Mozilla Firefox or Safari.
- Common web applications include e-commerce shops, webmail, and social networking sites.

Topic 1.7: Application Types

- There are three basic types of mobile apps based on the technology used to develop the:
- **Native apps** are created for one specific platform or operating system.
- **Web apps** are responsive versions of websites that can work on any mobile device or OS because they're delivered using a browser.

- **Hybrid apps** are combinations of both native and web apps, but wrapped within a native app, giving it the ability to have its icon or be downloaded from an app store.

Topic 1.8: Native Apps

- Developed specifically for a particular mobile device
- Installed directly onto the device itself
- Needs to be downloaded via app stores such as
    - Apple App Store
    - Google Play store, etc.
- Built for specific mobile operating system such as
    - Apple iOS
    - Android OS
- An app made for Apple iOS will not work on Android OS or Windows OS
- Need to target all major mobile operating systems

require more money and more effort

- Pros
    - Can be Used offline - faster to open and access anytime
    - Allow direct access to device hardware that is either more difficult or impossible with web apps
    - Allow the user to use device-specific hand gestures
    - Full access to all device features and APIs.
    - Gets the approval of the app store they are intended for
    - User can be assured of improved safety and security of the app
- Cons
    - More expensive to develop - separate app for each target platform
    - The cost of app maintenance is higher - especially if this app supports more than one mobile platform
    - Getting the app approved for the various app stores can prove to be a long and tedious process
    - Needs to download and install the updates to the apps ontouser’ss mobile device
- Native apps are built specifically for one platform using its native programming language and tools.
- **iOS**:
    - **Language**: Swift, Objective-C
    - **Frameworks**: UIKit, SwiftUI
    - **Tools**: Xcode
    - **Example App**: Instagram (iOS)
- **Android**:
    - **Language**: Java, Kotlin
    - **Frameworks**: Android SDK
    - **Tools**: Android Studio
    - **Example App**: WhatsApp (Android)

Topic 1.9: Web Apps

- Internet-enabled applications

    - Accessible via the device's Web browser
- Don't need to be downloaded and installed onto a mobile device
- Written as web pages in HTML and CSS with interactive parts in JQuery, JavaScript, etc.
- Single web app can be used on most devices capable of surfing the web irrespective of the operating system

- Pros
    - Instantly accessible to users via a browser
    - Easier to update or maintain
    - Easily discoverable through search engines
    - Development is considerably more time and cost-effective than development of a native app
    - common programming languages and technologies
    - It has a much larger developer base.
- Cons
    - Only have limited scope as far as accessing a mobile device's features is concerned such as device-specific hand gestures, sensors, etc.
    - Many variations between web browsers and browser versions and phones
    - Challenging to develop a stable web app that runs on all devices without any issues
    - Not listed in 'App Stores'
    - Unavailable when offline, even as a basic version
- Web apps run in web browsers and are built using standard web technologies.
- **Languages**: HTML, CSS, JavaScript
- **Frameworks**: React, Angular, Vue.js
- **Tools**: Web browsers, developer tools
- **Example App**: Google Docs

Topic 1.10: Progressive Web Application

- A **progressive web app** (PWA) is an app that's built using web platform technologies
- Like a platform-specific app, it can be installed on the device.
- Offline and background operation
- Progressive web apps combine the best features of traditional websites and platform-specific apps
- PWAs offer a native app-like experience on the web, including offline capabilities and installation.
- **Languages**: HTML, CSS, JavaScript
- **Frameworks**: Workbox, Lighthouse (for auditing)
- **Tools**: Service Workers, Web App Manifests

Topic 1.11: Hybrid Apps

- Hybrid apps combine elements of both native and web applications. They are built using web technologies but run inside a native container.
- **Languages**: HTML, CSS, JavaScript
- **Frameworks**: Ionic, Apache Cordova, PhoneGap
- **Tools**: Web technologies wrapped in native code

- Part native apps, part web apps
- Like native apps,
    - available in an app store
    - can take advantage of some device features available
- Like web apps,
    - Rely on HTML, CSS , JS for browser rendering
- The heart of a hybrid mobile application is an application that is written with HTML, CSS, and JavaScript!
- Run from within a native application and its own embedded browser, which is essentially invisible to the user
    - iOS application would use the WKWebView to display the application
    - Android app would use the WebView element to do the same function.
- Pros
    - Don't need a web browser like web apps
    - Can access to a device's internal APIs and device hardware
    - Only one codebase is needed for hybrid apps
- Cons
    - Much slower than native apps
    - With hybrid app development, dependent on a third-party platform to deploy the app's wrapper
    - Customization support is limited.

Topic 1.12: Cross-platform Application

- Cross-platform app development refers to developing software that can run on multiple devices.
- Multi-platform compatibility is a pervasively desirable trait.
- Product to be available to as many consumers as possible.
- Cross-platform native apps use frameworks that allow development for multiple platforms from a single codebase, providing near-native performance.
- **Frameworks**: React Native, Flutter, Xamarin
- **Tools**: IDEs like Visual Studio, Android Studio
- **Example App**:
    - **React Native**: Facebook
    - **Flutter**: Google Ads
    - **Xamarin**: Microsoft Outlook

Topic 1.13: Native App vs Cross Platform

Native App Development	Cross Platform App Development
Native App Development is costly.	It is cost-effective.
Code cannot be reused	It supports code reusability. Same codebase is used across multiple platforms

Native apps are faster.	Cross Platform apps are slower than native apps

Topic 1.14: Self Reading

- Recommendation 1:https://railsware.com/blog/native-vs-hybrid-vs-cross-platform/
- Recommendation 2: https://litslink.com/blog/mobile-applications-development-native-web-cross-platform

2. Application Introduction

Topic 2.1: Agenda for CS2

- Cloud-Native Applications
- Serverless Applications
- Layered Architectures (client /server, 2/3 tier- N tier)
- Monolithic
- Distributed Architectures – Service-oriented architecture, Microservices

Topic 2.2: Cloud Native Application

- Cloud-native is an approach to developing, deploying, and running applications using modern methods and tools.
- The Cloud Native Computing Foundation(CNCF) defines

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

Topic 2.3: Why Cloud Native

- Monolithic Application
    - A monolithic architecture refers to a traditional software design approach where an entire application is built as a single, self-contained unit.
- Key characteristics of monolithic architecture include:
    - Single Codebase
    - Tight Coupling
    - Single Deployment Unit
    - Centralized Database
    - Development and Scaling Challenges
    - Longer Development Cycles
    - Limited Fault Isolation

Topic 2.4: Cloud-Enabled Solutions

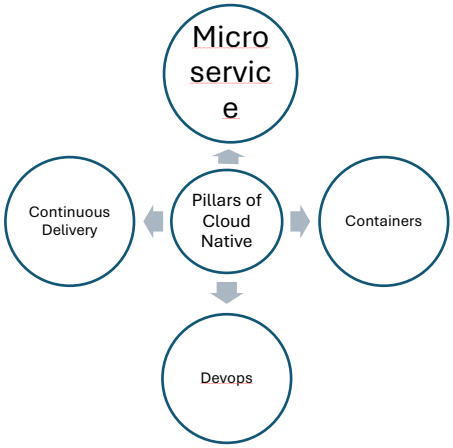
- What happens when you move a monolithic app from on-premises to cloud?
- A cloud-based application is an existing app shifted to a cloud ecosystem.
- They are not able to utilize the full potential of the cloud
    - Not scalable
    - Less automation
    - Longer Time to Market
- Cloud-Native Applications are designed to run on the cloud computing architecture.
- With cloud-native applications, you can take advantage of the automation and scalability that the cloud provides.

Topic 2.5: Cloud native vs. traditional applications

- A cloud enabled application is an application that was developed for deployment in a traditional data center but was later changed so that it could also run in a cloud environment
- Cloud-native applications, however, are built to operate only in the cloud.

- Developers design cloud-native applications to be
    - Scalable
    - platform agnostic
    - and comprised of microservices

Topic 2.6: Cloud Native Applications



Topic 2.7: Building Blocks

- Microservices
    - ✓ is an architectural approach to developing an application as a collection of small services
    - ✓ each service implements business capabilities, runs in its own process and communicates via HTTP APIs or messaging
- Containers
    - ✓ offer both efficiency and speed compared with standard virtual machines (VMs)
    - ✓ Using operating system (OS)-level virtualization, a single OS instance is dynamically divided among one or more isolated containers, each with a unique writable file system and resource quota
    - ✓ Low overhead of creating and destroying containers combined with the high packing density in a single VM makes containers an ideal compute vehicle for deploying individual microservices
- DevOps
    - ✓ Collaboration between software developers and IT operations to constantly deliver high-quality software that solves customer challenges
    - ✓ Creates a culture and an environment where building, testing, and releasing software happens rapidly, frequently, and more consistently
    - ✓ Continuous Delivery is about shipping small batches of software to production constantly through automation
    - ✓ makes the act of releasing reliable, so organizations can deliver frequently, at less risk, and get feedback faster from end users

Topic 2.8: Advantages

- Reduced Time to Market
- Ease of Management
- Scalability and Flexibility
- Reduced Cost

Topic 2.9: Serverless Application

- Serverless architecture is an approach to software design that allows developers to build and run services without managing the underlying infrastructure.
- Cloud service providers automatically provision, scale, and manage the infrastructure required to run the code.

Function as a Service

- One of the most popular serverless architectures is Function as a Service (FaaS)
- Developers write their application code as a set of discrete functions.
- Each function will perform a specific task when triggered by an event
- When a function is invoked, the cloud provider executes the function
- The execution process is abstracted away from the view of developers.
    - Examples:
        - AWS: AWS Lambda
        - Microsoft Azure: Azure Functions
        - Google Cloud: Cloud Functions

Topic 2.10: Serverless

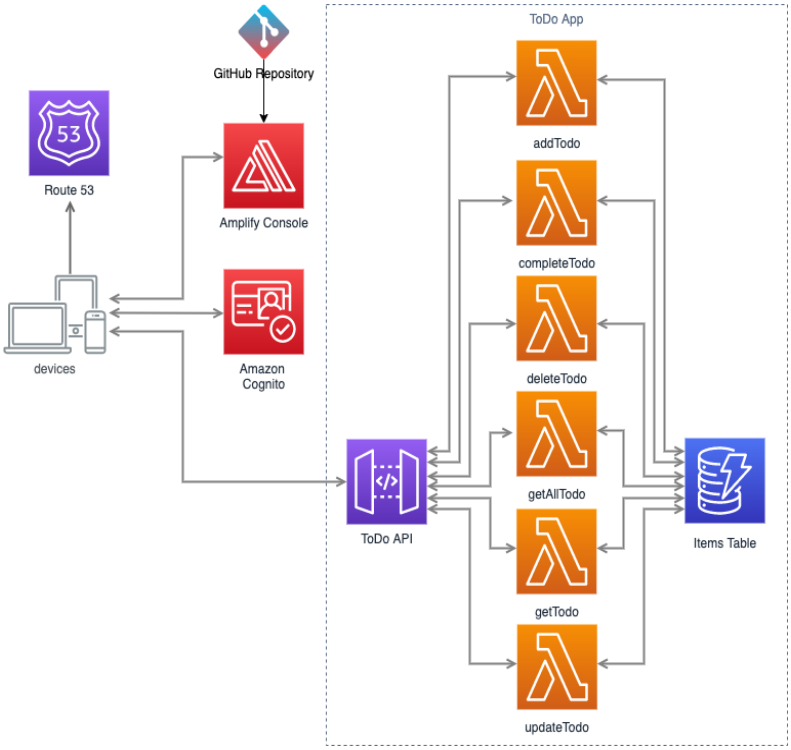
- Benefits
    - Reduced operational cost
    - Easier operational management
    - Scalability
- Drawbacks
    - Loss of control
    - Vendor lock-in
    - Multitenancy problems
    - Security concerns

Topic 2.11: AWS Serverless

- AWS provides a set of fully managed services that can be used to build and run serverless applications
- AWS Lambda
    - Allows to run code without provisioning or managing servers
- AWS Fargate
    - Purpose-built serverless compute engine for containers
- Amazon API Gateway
    - Fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale

Topic 2.12: Example: AWS Serverless Architecture

- General-purpose, event-driven, web application back-end that uses
- AWS Lambda, Amazon API Gateway for its business logic
- Uses Amazon DynamoDB as its database
- Uses Amazon Cognito for user management
- All static content is hosted using AWS Amplify Console


ToDo App
GitHub Repository
Route 53
Amplify Console
devices
Amazon Cognito
addTodo
completeTodo
deleteTodo
getAllTodo
getTodo
updateTodo
ToDo API
Items Table

3. Web Application

Topic 3.1: Activity

- Web site: https://whatismyipaddress.com/
- API : https://api.ipify.org/?format=json

Topic 3.2: Layered Pattern

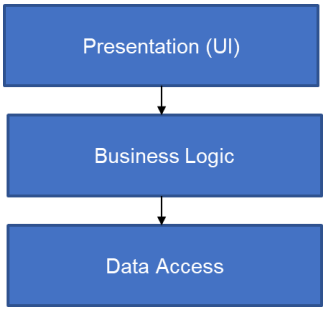
- Context: Separation of concern
- Problem: Modules of the system may be independently developed and maintained, supporting portability, modifiability, and reuse.
- **Solution**: The layered pattern divides the software into units called layers.
- Each layer is a grouping of modules that offers a cohesive set of services.
- Each partition is exposed through a interface.
- Layers interact according to a strict ordering relation and unidirectional.

Topic 3.3: Web Application

- Web application follows the Layered Architecture.
- Two-layer systems
  - Typical client-server system
  - The client held the user interface and other application code, and the server was usually a relational database.
  - Embed the logic directly into the UI screens
  - Alternative: put the domain logic in the database as stored procedures

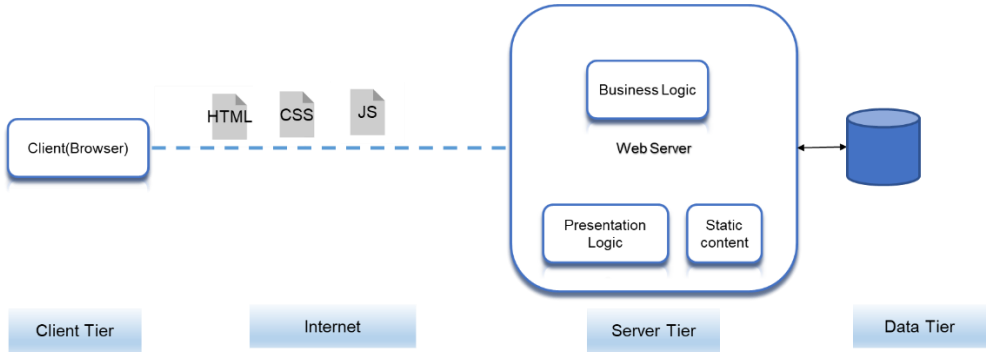
Topic 3.4: 3-Tier Architecture

- A typical Web Application:



Topic 3.5: Traditional Web Application

- Three Tier

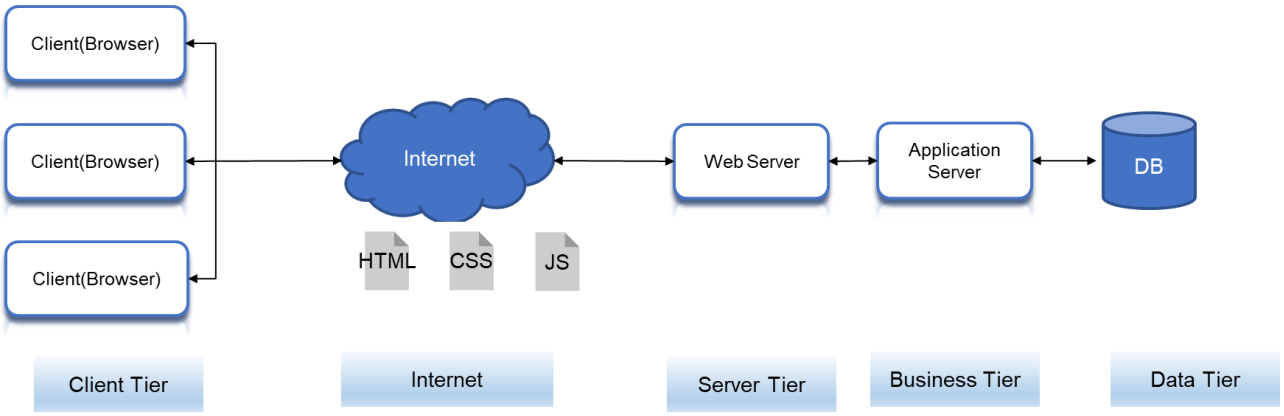


Topic 3.6: Ensuring a clear separation of concern

- **Scenario 1:** A list of products in which all the products that sold over 10 percent more than they did the previous month were colored in red.
- **Method 1:** (putting domain logic into the presentation)
- Developers placed logic in the presentation layer that compared this month's sales to last month's sales, and if the difference was more than 10 percent, they set the color to red
- **Method 2:**
- To properly separate the layers, you need a method in the domain layer to indicate if a product has improved sales. This method makes the comparison between the two months and returns a Boolean value.
- The presentation layer then calls this Boolean method and, if true, highlights the product in red.

- **Scenario 2:**You need to add different layers to an application, such as a command-line interface to a Web application.
- Method: If there's any functionality you have to duplicate to do this, that's a sign of where domain logic has leaked into the presentation
- Similarly, do you have to duplicate logic to replace a relational database with an XML file?

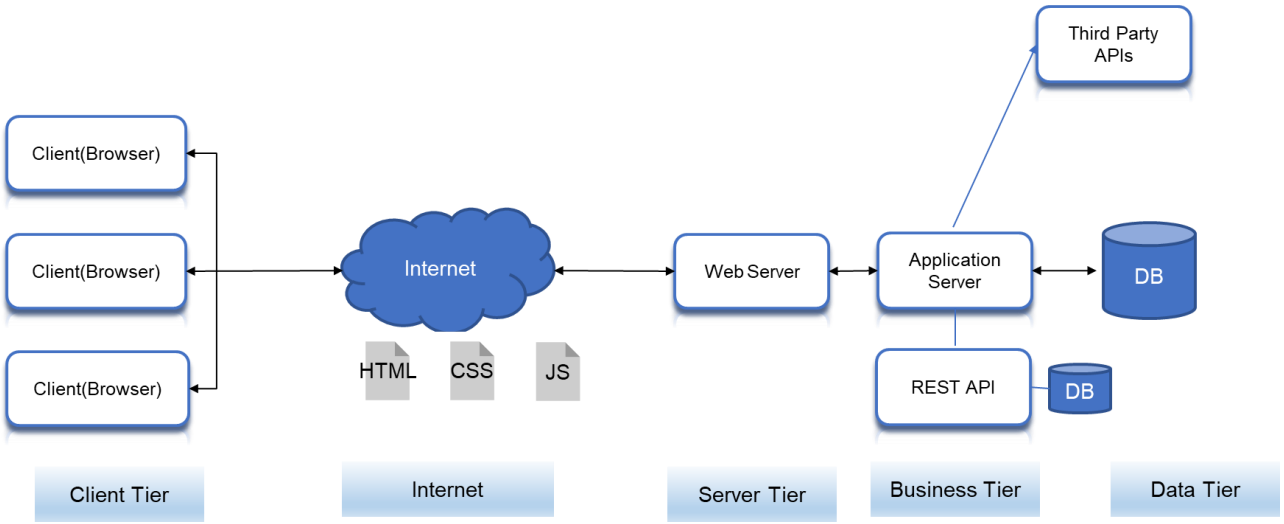
Topic 3.7: N-Tier Web Application



Topic 3.8: Characteristics

- Each tier is completely independent.
- The nth tier only has to know how to handle a request from the n+1th tier
- Tiers make it easier to ensure security and to optimize performance and availability in specialized ways.

Topic 3.9: Service Based Web Application



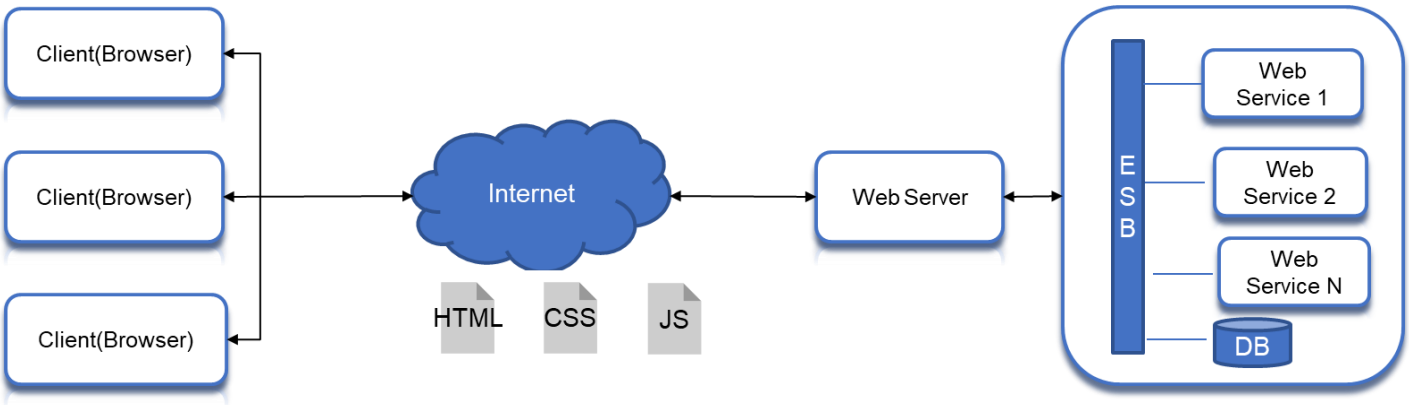
Topic 3.10: Monolithic Application

- The application is deployed as a single monolithic application.
- For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat
- Difficulty to scale
- Redeployment of the entire application in case of change

Topic 3.11: Service Oriented Architecture

- Service-oriented architecture (SOA) is a business-centric IT architectural approach that supports integrating your business as linked, repeatable business tasks or services.
- SOA exposes business functionalities as services to be consumed by applications.
- Services are
    - loosely coupled
    - Autonomous
    - **Robert C. Martin's Single Responsibility Principle**
- Gather together the things that change for the same reasons. Separate those things that change for different reasons.

Topic 3.12: SOA Based Web Application



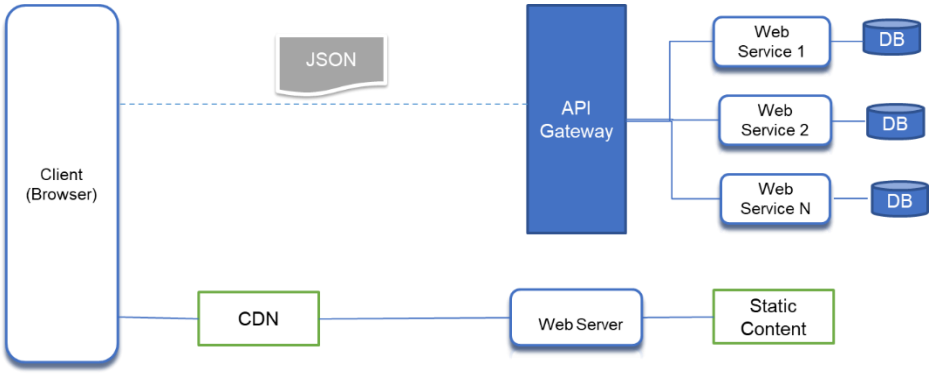
Topic 3.13: Microservice Architecture

- Microservice Application

- ✓ is composed of many small, independent services
- ✓ each service implements a single business capability
- ✓ are loosely coupled, communicating through API contracts
- ✓ can be built by a small, focused development team

- More complex to build and manage
  - ✓ requires a mature development and DevOps culture
  - ✓ can lead to higher release velocity, and a more resilient architecture

Topic 3.14: Microservices Application



Topic 3.15: API

- API stands for 'Application Programming Interface'
- API: an interface to access whatever resource it points to: data, server software, or other applications
- In an internet-connected world, web and mobile applications are designed for humans to use
- While APIs are designed for other digital systems and applications to use

Topic 3.16: What is a API?

- An API is a software intermediary that allows two applications to talk to each other
  - ✓ API is the messenger that delivers your request to the provider that you're requesting it from and then delivers the response back to you
- An API is independent of their respective implementations.
- Changing the underlying implementation ca be done without affecting the users
- APIs make it possible to integrate different systems together
  - ✓ like Customer Relationship Management systems, databases, or even school learning management systems
- In this metaphor, a customer is like a user, who tells the waiter what she wants
- The waiter is like an API,
  - ✓ receiving the customer's order
  - ✓ translating the order into easy-to-follow instructions that the kitchen then uses to fulfill that order—often following a specific set of codes
  - ✓ or input, that the kitchen easily recognizes
- The kitchen is like a server that creates the order in the manner the customer wants it, hopefully!
- When the food is ready, the waiter picks up the order and delivers it to the customer.
- Similarly, the API delivers the response.

Topic 3.17: Types

- Public API
    - Twitter API, Facebook API, Google Maps API, and more
    - Granting Outside Access to Your Assets
    - Provide a set of instructions and standards for accessing the information and services being shared
    - Making it possible for external developers to build an application around those assets
    - Much more restricted in the assets they share, given they're sharing them publicly with developers around the web
- Private API
    - Self-Service Developer & Partner Portal API
    - Far more common (and possibly even more beneficial, from a business standpoint)
    - Give developers an easy way to plug right into back-end systems, data, and software
    - Letting engineering teams do their jobs in less time, with fewer resources
    - All about productivity, partnerships, and facilitating service-oriented architectures

Topic 3.18: API Paradigm

- Over the years, multiple API paradigms have emerged such as
    - ✓ REST
    - ✓ RPC
    - ✓ GraphQL
    - ✓ WebHooks
    - ✓ and WebSockets
- Broadly can be classified as
    - ✓ Request-Response APIs
    - ✓ Event Driven APIs

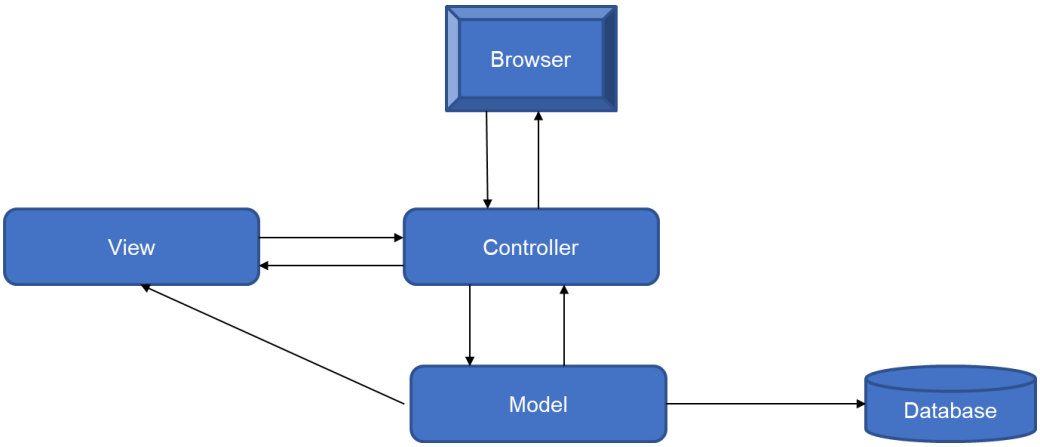
Topic 3.19: Request-Response APIs

- Typically expose an interface through a HTTP-based web server
- APIs define a set of endpoints
    - ✓ Clients make HTTP requests for data to those endpoints
    - ✓ Server returns responses
- The response is typically sent back as JSON or XML
- Three common paradigms used to expose request–response APIs:
    - ✓ REST
    - ✓ RPC
    - ✓ and GraphQL

Topic 3.20: Model View Controller (MVC)

- Software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements
- Supported well in JavaScript, Python, Ruby, PHP, Java, C#
- Three main logical components:

    - the **model**,
    - the **view**, and
    - the **controller**.
- Separates functionality
- Promotes organized programming
- Separating the presentation from the model and
- Separating the controller from the view.



- Data related Logic
- Interacts with the Database
- Communicated with controller
- Updates view (on few Frameworks)
- User Interface
- Communicates with controller
- Template engines
- Implementing the view in MVC
    - Template View and
        - *Renders information into HTML by embedding markers in an HTML page.*
    - Transform View
        - writing a program that looks at domain-oriented data and converts it to HTML.
- Receives input(from view)
- Process request(GET, POST)
- Gets data from the model
- Pass data to view
- Load a plain view(static HTML pages)
- Implementing Controller
    - Page controller
    - Front controller
    - Application controller

Topic 3.21: Example

- /routes

- Users/profiles/id = Users.getUser(id)

- /controllers

```
Class Users{
function getUser(id){
Profile=this.UserModel.getUser(id)
renderView("/users/profiles", profile)
}
}
```

- /Models

```
Class UserModel{
getUser(id){ sql query to db , return data}
}
```

- /views

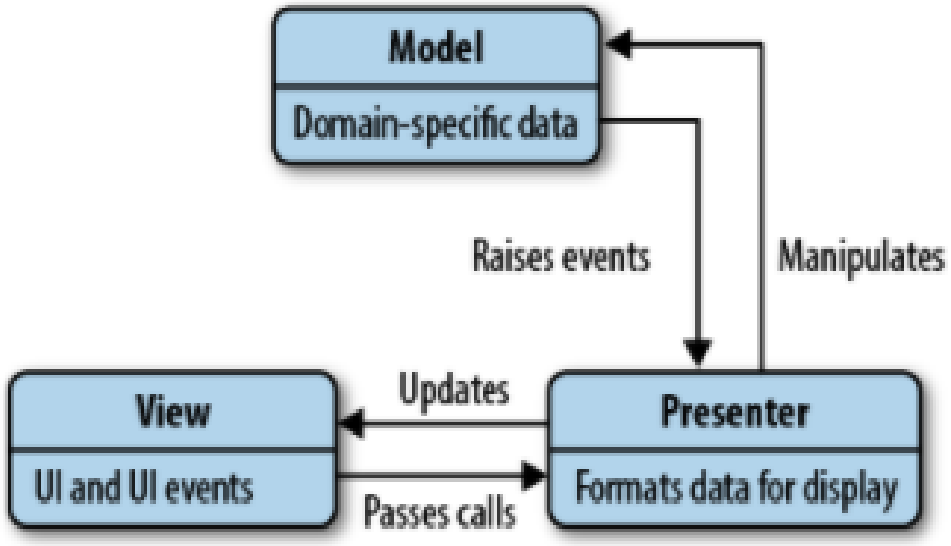
/users

/profile

```
<h1> {{profile.name}}</h1>
<ul>
<li>email : {{profile.email}}</li>
<li>Phone: {{profile:phone}}</li>
</ul>
```

Topic 3.22: Model View Presenter

- The view is a passive interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data.



Topic 3.23: MTV (Model, Template, and View)

- This Model similar to MVC acts as an interface for your data

- The View executes the business logic and interacts with the Model and renders the template.
- It accepts HTTP request and then return HTTP responses.
- The Template is the component which makes MVT different from MVC.
- Templates act as the presentation layer and are basically the HTML code that renders the data.
- The content in these files can be either static on dynamic.

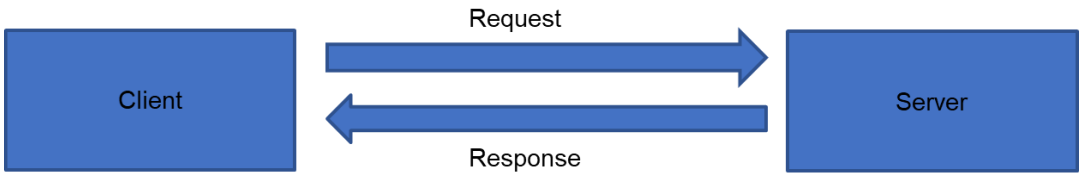
4. HTTP

Topic 4.1: Client Server Pattern

- **Context:** shared resources and services accessed by distributed clients
- **Problem:** By managing a set of shared resources and services, promote modifiability and reuse
- **Solution:** Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers.

Topic 4.2: Client Server Architecture

- Request – Response Model
- Providers of a resource or service, Server
- Service requester/Consumer - Client



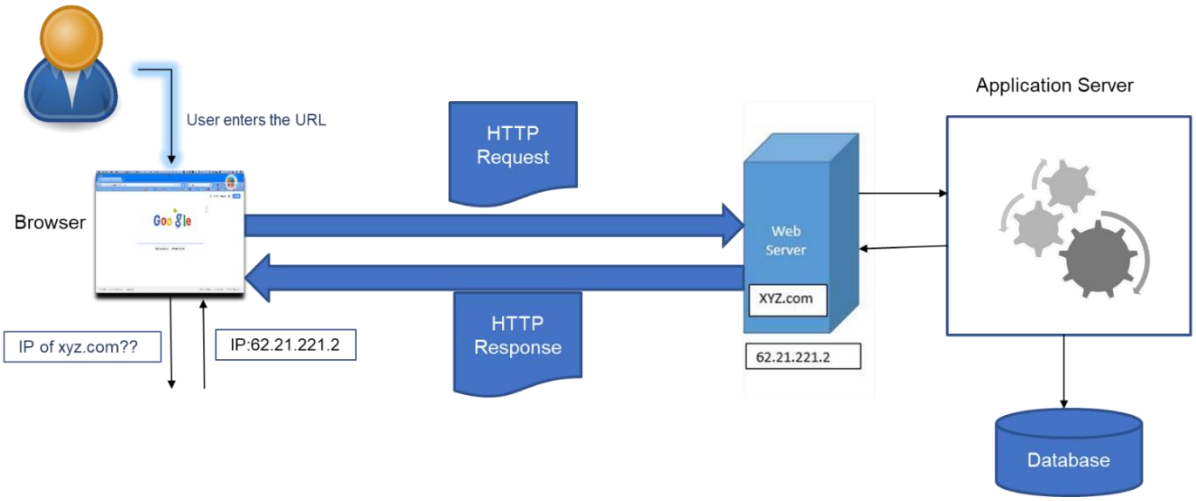
Topic 4.3: Client Server Architecture

- **Benefits**
  - ✓ **Higher security**
  - ✓ **Centralized data access**.
  - ✓ **Ease of maintenance**.
- The client-server architectural style has evolved into the more general 3-tier (or N-tier) architectural style for the web

Topic 4.4: Variants of Client Server Pattern:

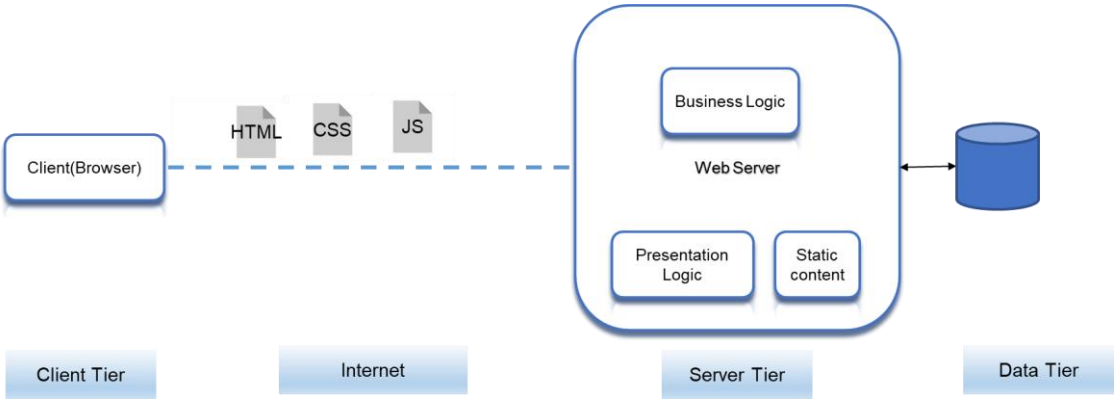
- Peer-to-Peer (P2P) applications
- Application servers
- Variations
  - ✓ Web browsers don't block until the data request is served up - Asynchronous
  - ✓ In some client-server patterns, servers can initiate certain actions on their clients.
  - ✓ Service calls over a request/reply connector are bracketed by a "session"

Topic 4.5: Working of the Web

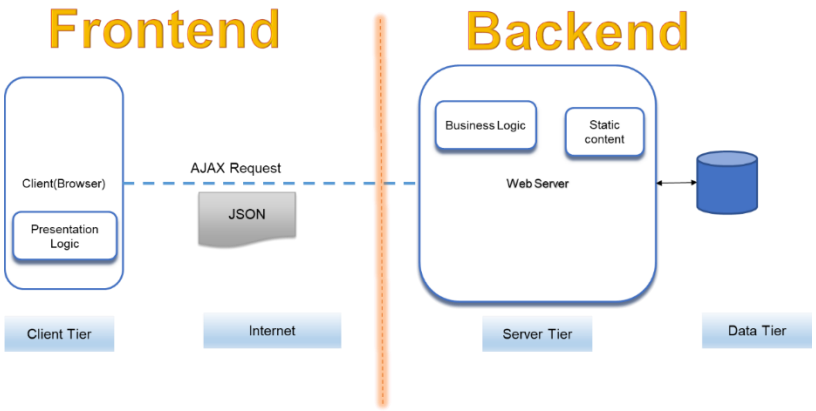


Topic 4.6: Traditional Web Application

- Three Tier



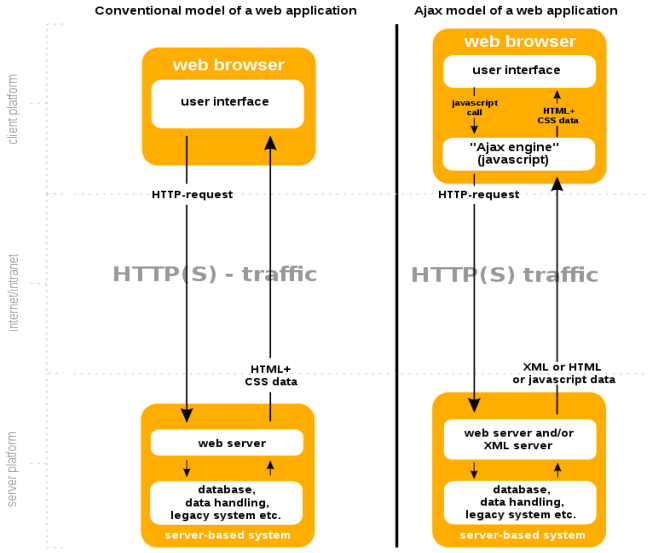
Topic 4.7: Modern Web Application



Topic 4.8: AJAX

- Asynchronous JavaScript and XML (Ajax) refer to a concept that is used to develop web applications in a better way.
- Ajax defines a method of initiating client-to-server communication without page reloads.
- It provides a way to enable partial page updates.
- In an Ajax web application, the user is not interrupted in interactions with the web application.

Topic 4.9: Model of a Web application



Topic 4.10: Front-end Responsibilities

- User Interface elements
- Mark-up and web languages such as HTML, CSS, JavaScript and supporting libraries
- Asynchronous request handling and AJAX
- Single-page applications (with frameworks like React, AngularJS or Vue.js)
- Web performance
- Responsive web design
- Cross-browser compatibility issues and workarounds
- End-to-end testing with a headless browser
- Build automation to transform and bundle JavaScript files, reduce image size
- Search engine optimization
- Accessibility concerns

Topic 4.11: Backend Responsibilities

- Software Architecture
- Application Business Logic
- Application Data Access
- Database management
- Scripting languages like JavaScript, Node.js, PHP, Python, Ruby, Java etc.
- Automated testing frameworks for the language being used
- Scalability
- High availability
- Security concerns, authentication and authorization

Topic 4.12: Parts of an URL

- http://abc.company.com:80/a/b/c.html?user=John&year=2020#p2
- The scheme identifies the protocol used to fetch the content.
- Host name name of a machine to connect to.
- The server's port number allows multiple servers to run on the same machine.
- The hierarchical portion is used by the server to find content.
- The Query parameters provide additional parameters
- Fragment : Have the browser scroll the page to a specific part of the webpage fragment

Topic 4.13: Different types of links

- Full URL: News
- Absolute URL: <a href="/stock/quote.html">
    - same as http://www.xyz.com/stock/quote.html
- Relative URL (intra-site links): <a href="b/March.html">
    - same as http://www.xyz.com/news/2008/March.html
- Define an anchor point (a position that can be referenced with # notation):
    - <a id="sec3">

- Go to a different place in the same page: <a href="#sec3">

Topic 4.14: URL Encoding

- How are special characters sent in the URL?
    - http://www.xyz.com/companyInfo?name=A&B CO
    - Any character in a URL other than A-Z, a-z, 0-9, or any of -_.~ must be represented as %xx, where xx is the hexadecimal value of the character:
    - http://www.xyz.com/companyInfo?name=A%26B%20CO
- Escaping is a commonly used technique.

Topic 4.15: Domain Name System

- The Domain Name System (DNS) is the phonebook of the Internet.
- Humans access information online through domain names such as google.com or facebook.com.
- The network of devices interacts through Internet Protocol (IP) addresses.
- DNS translates domain names to IP addresses so browsers can load Internet resources.
- Each device connected to the Internet has a unique IP address, which other machines use to find the device.
- DNS servers eliminate the need for humans to memorize IP addresses

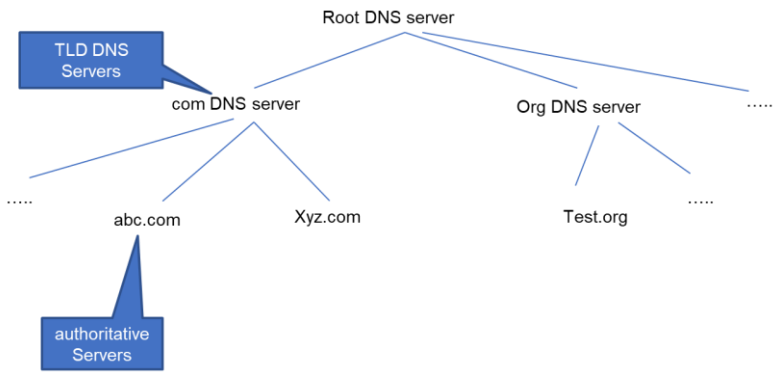
Topic 4.16: Domain Name System

- The Domain Name System resolves the names of internet sites with their underlying IP addresses.
- A DNS server is a computer server that contains a database of public IP addresses and their associated hostnames.
- DNS is a distributed database implemented in a hierarchy of name servers.
- It is an application layer protocol for message exchange between clients and servers.

Topic 4.17: Domain Name System

- The **DNS recursor** is a server designed to receive queries from client machines through applications such as web browsers.
- The **root server** is the first step in translating (resolving) human-readable host names into IP addresses.
- TLD nameserver - The top level domain server (TLD)
- The authoritative nameserver is the last stop in the nameserver query

Topic 4.18: Domain Name System Lookup



Topic 4.19: Content Delivery Network

- A content delivery network (CDN) is a geographically distributed group of servers that caches content close to end users.

- A CDN allows for the quick transfer of assets needed for loading Internet content, including HTML pages, JavaScript files, stylesheets, images, and videos.
- **Is a CDN the same as a web host?**
- CDN can't replace the need for proper web hosting, It improve Web performance!!

- A CDN improves website load times.
- The globally distributed nature of a CDN reduces the distance between users and website resources.
- A CDN caches content (such as images, videos, or webpages) in proxy servers that are located closer to end users than origin servers.
- CDNs also reduce the amount of data transferred by reducing file sizes using minification and file compression tactics.
- Load balancing distributes network traffic evenly across several servers, making it easier to scale rapid boosts in traffic.

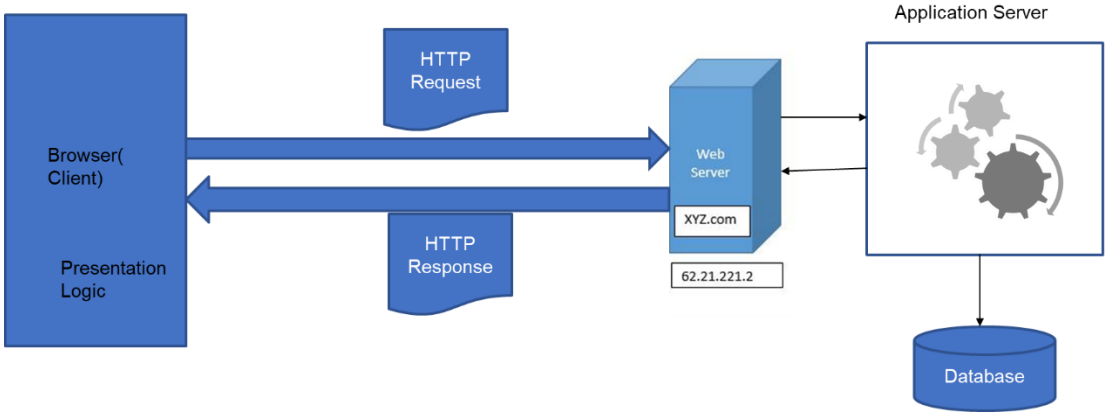
Topic 4.20: Benefits of using a CDN

- Improving website load times
- Reducing bandwidth costs
- Increasing content availability and redundancy
- Improving website security

Topic 4.21: Client-side rendering (CSR)

- Client-side rendering (CSR) is a technique for rendering web content on the client-side, i.e., in the user's browser.
- With CSR, the client requests a minimal HTML file from the server containing the necessary JavaScript and CSS files.
- When the client loads the JavaScript files, the JavaScript code is executed, which renders the content in the browser.

Topic 4.22: Client Side rendering



Topic 4.23: Server-Side Rendering

- Server-side rendering (SSR) is a technique for rendering web content on the server-side, i.e., before the page is sent to the client.
- Server-Side Rendering is also named Pre-Rendering because the fetching of external data and transformation of components, content, and data into HTML happens before the result is sent to the client.
- There are several benefits of using server-side rendering:
    - Better SEO

- Faster Initial Page Load
- Improved Accessibility
- Better Performance on Low-Powered Devices

Topic 4.24: Static Site Generation

- Static site generation (SSG) is a popular approach to website development that involves generating a website's content ahead of time and delivering it as static HTML files to end-users.
- In SSG, the HTML is generated once, at build time.
- The HTML is stored in a CDN or elsewhere and re-used for each request.
- A static site generator combines the content and templates into a collection of static HTML files.
- This process may also include optimizing images, minifying code, and generating metadata.

Topic 4.25: Static Site Generation

- Benefits:
  - Performance
  - SEO
  - Cost
- Limitations:
  - Not suitable for all types
  - No Real-time data
- Some popular static site generators include Jekyll, Hugo, and Gatsby.

Topic 4.26: Client-Side Programming

- Involves everything users see on their screens.
- Major frontend technology stack components:
  - HTML, CSS and JS
- Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS)
  - HTML tells a browser how to display the content of web pages
  - CSS styles that content
  - Bootstrap is a helpful framework for managing HTML and CSS
- JavaScript (JS)
  - Makes web pages interactive
  - Many JavaScript libraries (such as jQuery, React.js)
  - frameworks (such as Angular, Vue, Backbone, and Ember)

Topic 4.27: Server-Side Programming

- Major server-side technology stack components:
  - Programming language, Framework, web server and databases
- Server-side programming languages used to create the logic of applications
- Frameworks offer lots of tools for simpler and faster development of applications
- Options
  - Ruby (Ruby on Rails)
  - Python (Django, Flask, Pylons)

- PHP (Laravel)
- Java (Spring)
- Scala (Play)
- Javascript (Express Node.js)

- Storage
  - Apps needs a place to store its data
  - Two types of databases:
    - relational and non-relational
  - Most common databases for web development:
    - MySQL (relational)
    - PostgreSQL (relational)
    - MongoDB (non-relational, document)
- Caching system
  - Used to reduce the load on the database and to handle large amounts of traffic
  - Memcached and Redis are the most widespread.
  - Web servers/Load balancers/Proxy servers
  - Needs a server to handle requests from clients' computers
  - Two major players: Apache, Nginx
  - Cloud Based Servers (EC2, Serverless, ELB)

Topic 4.28: Example TechStack

- MEAN / MERN / MEVN Stack
  - MongoDB: A NoSQL database that stores data in a flexible, JSON-like format.
  - Express.js: A minimal and flexible Node.js web application framework that provides robust features for web and mobile applications.
  - Frontend Framework
    - Vue.js: A JavaScript framework for building user interfaces.
    - Angular: A TypeScript-based open-source front-end web application framework maintained by Google.
    - React: A JavaScript library for building user interfaces, developed by Facebook.
  - Node.js: A JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to use JavaScript for server-side scripting.
- Python Stack:
  - Flask or Django: Flask is a lightweight Python web framework, and Django is a high-level Python web framework.
  - SQLAlchemy: An SQL toolkit and Object-Relational Mapping (ORM) library.
  - Django REST framework: If using Django for building APIs.
  - Database: Various options, including SQLite, PostgreSQL, or MySQL.
  - HTML/CSS/JavaScript or React/Angular for frontend

Topic 4.29: References

- https://developer.chrome.com/docs/devtools/

- https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction
- Chapter 1: Full Stack Web Development: The Comprehensive Guide by Philip Ackermann Shroff/Rheinwerk Computing; First Edition (2 August 2023)

5. CS4.1 HTTP

- HTTP
    - HTTP Request- Response and its structure
    - HTTP Methods;
    - HTTP Headers
    - Connection management - HTTP/1.1 and HTTP/2
- Synchronous and asynchronous communication
- Communication with Backend
    - AJAX, Fetch API
    - Webhooks
    - Server-Sent Events
- Polling
- Bidirectional communication - Web sockets

Topic 5.1 HTTP

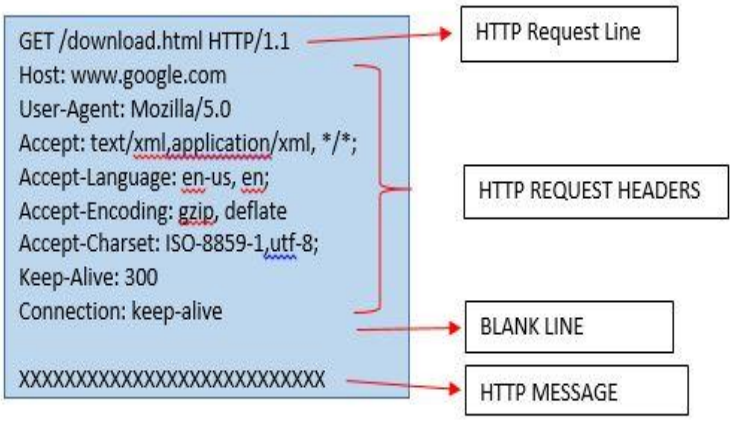
Topic 5.2 HyperText Transfer Protocol(HTTP)

- HTTP is a request-response client-server protocol
- HTTP is a stateless protocol.
- HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems.
- Each HTTP communication (request or response) between a browser and a Web server consists of two parts:
    - A header and
    - A body
- The header contains information about the communication.
- The body contains the data of the communication (optional).

Topic 5.3 HTTP Request

- The general form of an HTTP request is:

1. HTTP Request Line
2. Header fields
3. Blank line
4. Message body (Optional)



Topic 5.4 Request Line

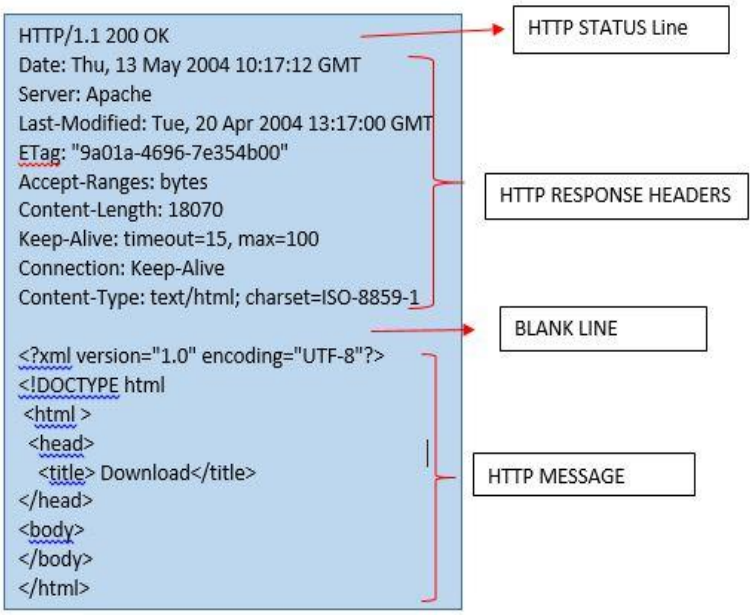
- The first line of the header is called the request line:
- **request-method-name /request-URI /HTTP-version**
- Examples of request line are:
- GET /test.html HTTP/1.1
- HEAD /query.html HTTP/1.0
- POST /index.html HTTP/1.1

Topic 5.5 Request Headers

- The request headers are in the form of name:value pairs. Multiple values, separated by commas, can be specified.
- request-header-name: request-header-value1, request-header-value2, ...
- Examples of request headers are:
- Host: www.xyz.com
- Connection: Keep-Alive
- Accept: image/gif, image/jpeg, */*
- Accept-Language: us-en, fr, cn

Topic 5.6 HTTP Response

- The general form of an HTTP response is:
- 1. Status line
- 2. Response header fields
- 3. Blank line
- 4. Response body



Topic 5.7 Status Line

- The first line is called the status line.
- HTTP-version status-code reason-phrase
- status-code: a 3-digit number generated by the server to reflect the outcome of the request.
- reason-phrase: gives a short explanation to the status code.
- Common status code and reason phrase are "200 OK", "404 Not Found", "403 Forbidden", "500 Internal Server Error".
- Examples of status line are:
- HTTP/1.1 200 OK
- HTTP/1.0 404 Not Found
- HTTP/1.1 403 Forbidden

Topic 5.8 STATUS CODE

- Status code is a three-digit number; first digit the specifies the general status
- 1 => Informational
- 2 => Success
- 3 => Redirection
- 4 => Client error
- 5 => Server error

Topic 5.9 Common HTTP Response Status Codes

- 200 OK Indicates a nonspecific success
- 201 Created Sent primarily by collections and stores but sometimes also by controllers, to indicate that a new resource has been created
- 204 No Content Indicates that the body has been intentionally left blank
- 301 Moved Permanently Indicates that a new *permanent* URI has been assigned to the client's requested resource

- 303 See Other Sent by controllers to return results that it considers optional
- 401 Unauthorized Sent when the client either provided invalid credentials or forgot to send them
- 402 Forbidden Sent to deny access to a protected resource
- 404 Not Found Sent when the client tried to interact with a URI that the REST API could not map to a resource
- 405 Method Not Allowed Sent when the client tried to interact using an unsupported HTTP method
- 406 Not Acceptable Sent when the client tried to request data in an unsupported media type format
- 500 Internal Server Error Tells the client that the API is having problems of its own

Topic 5.10 Response Headers

- The response headers are in the form name:value pairs:
- response-header-name: response-header-value1, response-header-value2, ...
- Examples of response headers are:
- Content-Type: text/html
- Content-Length: 35
- Connection: Keep-Alive
- Keep-Alive: timeout=15, max=100
- The response message body contains the resource data requested.

Topic 5.11 HTTP Methods (Verbs)

- GET - Fetch a URL
- HEAD - Fetch information about a URL
- PUT - Store to an URL
- POST - Send form data to a URL and get a response back
- DELETE - Delete a resource in URL
- GET and POST (forms) are commonly used.

Topic 5.12 HTTP Methods

- HTTP defines a set of **request methods** to indicate the desired action for a given resource.
- The request methods are sometimes referred to as *HTTP verbs*.
- GET - Fetch a URL
- HEAD - Fetch information about a URL
- PUT - Store to an URL
- POST - Send form data to a URL and get a response back
- DELETE - Delete a resource in URL
- GET and POST (forms) are commonly used.

Topic 5.13 HTTP Methods GET

- The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
- The GET request method is said to be a safe operation, which means it should not change the state of any resource on the server.
- The GET method is used to request any of the following resources:

- A webpage or HTML file.
- An image or video.
- A JSON document.
- A CSS file or JavaScript file.
- An XML file.

Topic 5.14 HTTP Methods POST

- The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.
- The POST HTTP request method sends data to the server for processing.
- The data sent to the server is typically in the following form:
    - Input fields from online forms.
    - XML or JSON data.
    - Text data from query parameters.
- A POST operation is not considered a safe operation, as it has the power to update the state of the server and cause potential side effects to the server's state when executed.
- The HTTP POST method is not required to be idempotent either, which means it can leave data and resources on the server in a different state each time it is invoked.

Topic 5.15 HTTP Methods HEAD

- The HEAD method requests a response identical to a GET request but without the response body.
- The HTTP HEAD method returns metadata about a resource on the server.
- The HTTP HEAD method is commonly used to check the following conditions:
    - The size of a resource on the server.
    - If a resource exists on the server or not.
    - The last-modified date of a resource.
    - Validity of a cached resource on the server.
    - The following example shows sample data returned from a HEAD request:

HTTP/1.1 200 OK

Date: Fri, 19 Aug 2023 12:00:00 GMT

Content-Type: text/html

Content-Length: 1234

Last-Modified: Thu, 18 Aug 2023 15:30:00 GMT

Topic 5.16 HTTP Methods PUT

- The HTTP PUT method is used to replace a resource identified with a given URL completely.
- The HTTP PUT request method includes two rules:
- A PUT operation always includes a payload that describes a completely new resource definition to be saved by the server.
- The PUT operation uses the exact URL of the target resource.
- If a resource exists at the URL provided by a PUT operation, the resource's representation is completely replaced.
- A new resource is created if a resource does not exist at that URL.
- The payload of a PUT operation can be anything that the server understands, although JSON and XML are the most common data exchange formats for RESTful web services.

Topic 5.17 HTTP Methods DELETE

The DELETE method deletes the specified resource.

CONNECT

The CONNECT method establishes a tunnel to the server identified by the target resource.

OPTIONS

The OPTIONS method describes the communication options for the target resource.

TRACE

The TRACE method performs a message loop-back test along the path to the target resource.

PATCH

The PATCH method applies partial modifications to a resource.

Topic 5.18 HTTP Request Methods

HTTP Request methods		
	SAFE	**IDEMPOTENT**
GET	Yes	Yes
POST	No	No
PUT	No	Yes
PATCH	No	No
DELETE	No	Yes
TRACE	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
CONNECT	No	No

Topic 5.19 HTTP HEADERS

- HTTP headers allow the client and the server to pass additional information with the request or the response.
- An HTTP header consists of its name followed by a colon ':', then by its value.
- Headers can be grouped according to their contexts:
- General header: Headers applying to both requests and responses but with no relation to the data eventually transmitted in the body.
- Request header: Headers containing more information about the resource to be fetched or about the client.
- Response header: Headers with additional information about the response, like its location or about the server itself (name and version etc.).
- Entity header: Headers containing more information about the body of the entity, like its content length or its MIME-type.

Topic 5.20 HTTP Headers -Content

- The Accept header lists the MIME types of media resources that the agent is willing to process.
- Each combined with a quality factor, a parameter indicating the relative degree of preference between the different MIME types.

- A media type also known as MIME type is a two-part identifier for file formats and format contents transmitted on the Internet.
- Form: type/subtype
- Examples: text/plain, text/html, image/gif, image/jpeg
- Accept: image/gif, image/jpeg, */*

Topic 5.21 The Accept-Charset header

- It indicates to the server what kinds of character encodings are understood by the user-agent.
- Accept-Charset:utf-8

Topic 5.22 HTTP Headers -Content

Topic 5.23 The Accept-Encoding header

- The Accept-Encoding header defines the acceptable content-encoding (supported compressions).
- The value is a q-factor list (e.g.: br, gzip;q=0.8) that indicates the priority of the encoding values.
- Compressing HTTP messages is one of the most important ways to improve the performance of a Web site.
- Accept-Encoding:gzip, deflate

Topic 5.24 The Accept-Language header

- It is used to indicate the language preference of the user.
- Accept-Language: en-us

Topic 5.25 The User-Agent header

It identifies the browser sending the request.

Topic 5.26 HTTP Headers- Caching

- The Cache-Control general-header field is used to specify directives for caching mechanisms in both requests and responses.
- Caching directives are unidirectional, i.e., directive in a request is not implying that the same directive is to be given in the response.
- Standard Cache-Control directives that can be used by the client in an HTTP request.
    - Cache-Control: max-age=<seconds>
    - Cache-Control: no-cache
    - Cache-Control: no-store
    - Cache-Control: no-transform
- Standard Cache-Control directives that can be used by the server in an HTTP response.
    - Cache-Control: must-revalidate
    - Cache-Control: no-cache
    - Cache-Control: no-store
    - Cache-Control: no-transform
    - Cache-Control: public
    - Cache-Control: private

Topic 5.27 HTTP Headers- Caching

- The Expires header contains the date/time after which the response is considered stale.
- Invalid dates, like the value 0, represent a date in the past and mean that the resource is already expired.

- If there is a Cache-Control header with the "max-age" directive in the response, the Expires header is ignored.
- Expires: Wed, 21 Oct 2015 07:28:00 GMT

Topic 5.28 HTTP Headers -Caching

- The ETag HTTP response header is an identifier for a specific version of a resource.
- If the resource at a given URL changes, a new Etag value must be generated.
- ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4“
- The client will send the Etag value of its cached resource along in an If-None-Match header field:
- If-None-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4“
- The server compares the client's ETag (sent with If-None-Match) with the ETag for its current version of the resource
- If both values match, the server send back a 304 Not Modified status, without any body
- Tells the client that the cached version of the response is still good.

Topic 5.29 HTTP Headers-Caching

- The Last-Modified response HTTP header contains the date and time at which the origin server believes the resource was last modified.
- It is used as a validator to determine if a resource received or stored is the same.
- Less accurate than an ETag header

Topic 5.30 Redirection

- In HTTP, a redirection is triggered by the server by sending special responses to a request: redirects.
- HTTP redirects are responses with a status code of 3xx.
- A browser, when receiving a redirect response, uses the new URL provided in the location header.
- Location: /index.html
- Permanent redirections
- 301 Moved Permanently
- Temporary Redirections
- 302 Temporary Redirect

Topic 5.31 HTTP Headers-Cookies

- An HTTP cookie is a small piece of data that a server sends to the user's web browser.
- The browser may store it and send it back with the next request to the same server.
- The Set-Cookie HTTP response header sends cookies from the server to the user agent.
- Set-Cookie: <cookie-name>=<cookie-value>
- The Cookie HTTP request header contains stored HTTP cookies previously sent by the server with the Set-Cookie header.
- Cookie: name=value; name2=value2; name3=value3

Topic 5.32 Summary

- HTTP Request response
- HTTP Methods
- HTTP Headers

6. CS 4.2 HTTP

Topic 6.1 Module 3: Web Protocols

- HTTP
    - HTTP Request- Response and its structure
    - HTTP Methods;
    - HTTP Headers
    - Connection management - HTTP/1.1 and HTTP/2
- Synchronous and asynchronous communication
- Communication with Backend
    - AJAX, Fetch API
    - Webhooks
    - Server-Sent Events
- Polling
- Bidirectional communication - Web sockets

Topic 6.2 Agenda

- Connection management - HTTP/1.1 and HTTP/2
- HTTPS

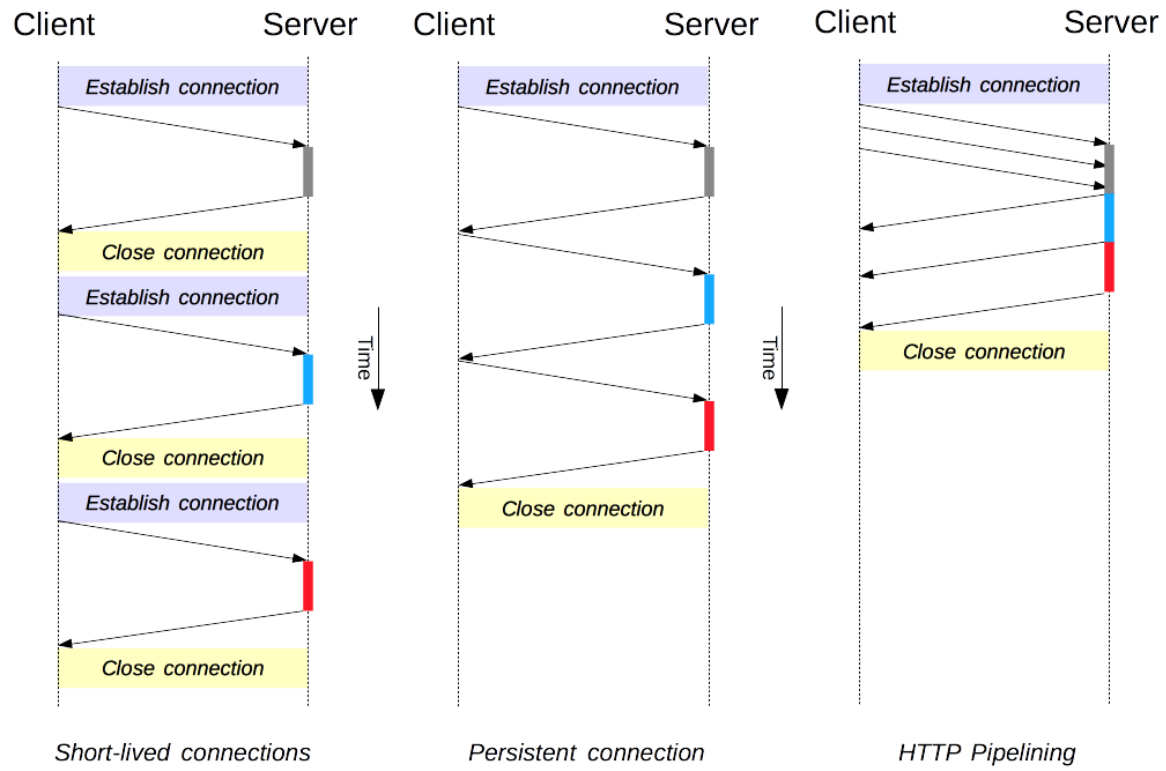
Topic 6.3 HTTP Headers- Connection Management

- The Connection general header controls whether or not the network connection stays open after the current transaction finishes.
- If the value sent is keep-alive, the connection is persistent and not closed, allowing for subsequent requests to the same server to be done.
- Connection: keep-alive
- Connection: close

Topic 6.4 Keep-alive Header

- The Keep-Alive general header allows the sender to hint about how the connection may be used to set a timeout and a maximum amount of requests.
- Keep-Alive: timeout=5, max=1000

Topic 6.5 HTTP Connection 1.x



Reference: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x

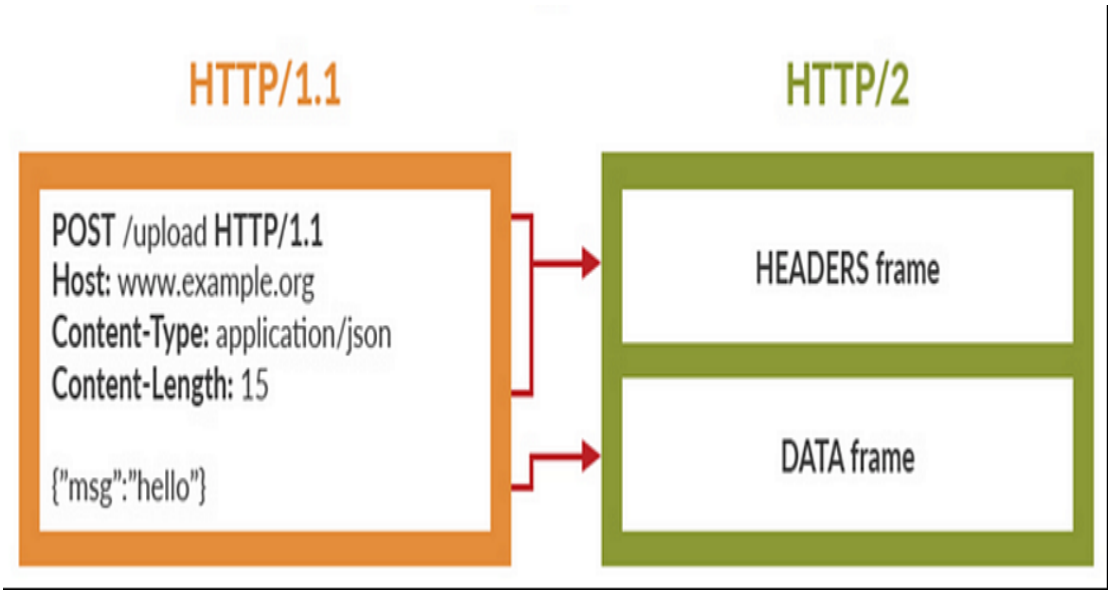
Topic 6.6 HTTP/1

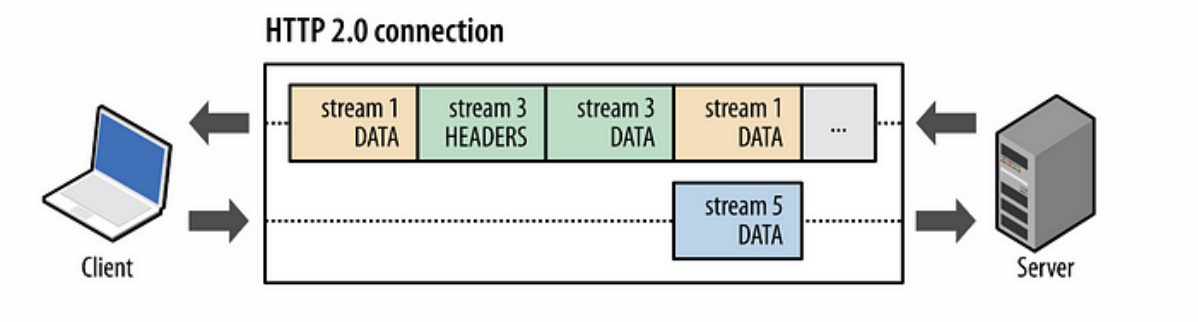
- HTTP/1.x clients need to use multiple connections to achieve concurrency and reduce latency;
- -> Head of line Problem
- HTTP/1.x does not compress request and response headers, causing unnecessary network traffic;
- HTTP/1.x does not allow effective resource prioritization, resulting in poor use of the underlying TCP connection

Topic 6.7 HTTP/2

- *HTTP/2 enables a more efficient use of network resources and a reduced perception of latency by introducing header field compression and allowing multiple concurrent exchanges on the same connection… Specifically, it allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance.*
- *The resulting protocol is more friendly to the network, because fewer TCP connections can be used in comparison to HTTP/1.x. This means less competition with other flows, and longer-lived connections, which in turn leads to better utilization of available network capacity. Finally, HTTP/2 also enables more efficient processing of messages through use of binary message framing.*

Topic 6.8 HTTP/2

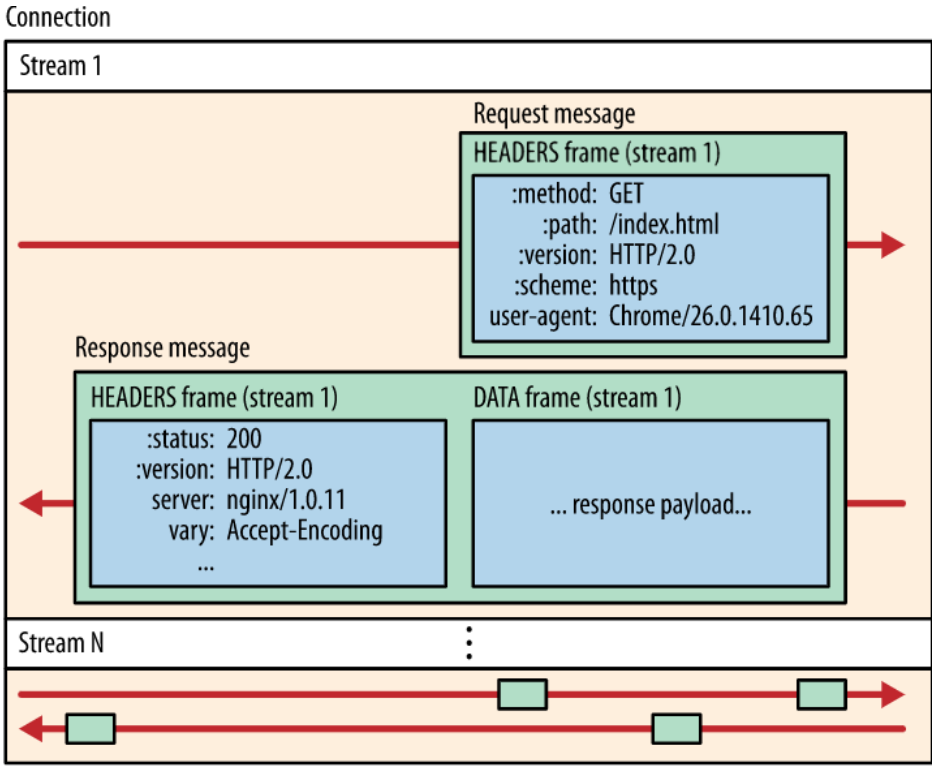




Reference: https://web.dev/performance-http2/

Topic 6.9 HTTP/2

- Frames -> Messages -> Streams -> A single TCP connection



Reference: https://web.dev/performance-http2/

Topic 6.10 HTTP/2

- The introduction of the new binary framing mechanism changes how the data is exchanged between the client and server.
- *Stream*: A bidirectional flow of bytes within an established connection, which may carry one or more messages.
- *Message*: A complete frame sequence that maps to a logical request or response message.
- *Frame*: The smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum, identifies the stream to which the frame belongs.

Topic 6.11 Features and Benefits of HTTP/2

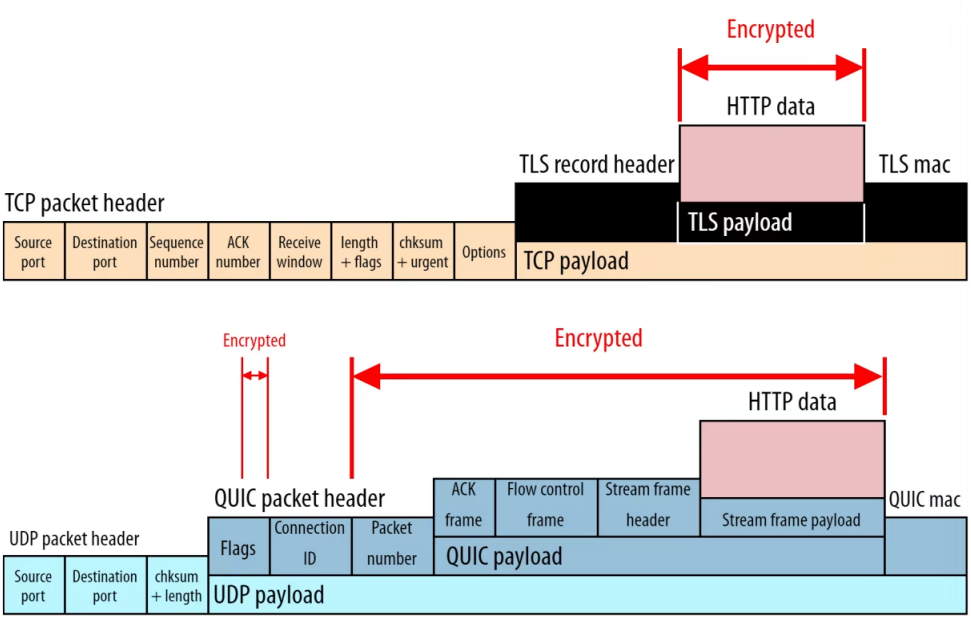
- It's a **binary** protocol
- It used header **compression** HPACK to reduce the overhead size
- It allows servers to "**push**" responses proactively into client caches instead of waiting for a new request for each resource
- It reduces additional **round trip times (RTT),** making your website load faster without any optimization.
- HTTP/2 supports response **prioritization**.

HTTP/2 also has problems, We now have HTTP/3 as well!

Topic 6.12 QUIC

- QUIC (Quick UDP Internet Connections) is a new encrypted-by-default Internet transport protocol.
- **Built-in security (and performance)**
- The initial QUIC handshake combines the typical TCP three-way handshake and the TLS 1.3 handshake.
- QUIC handshake only takes a single round-trip between client and server to complete
- It also encrypts additional connection metadata
- Reduced head-of-line blocking
- Improved congestion control

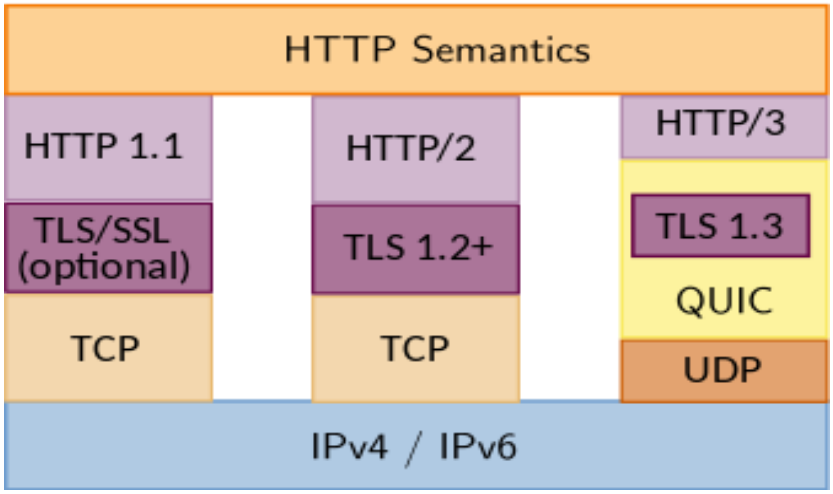
Topic 6.13 Connection Metadata – encrypted



Topic 6.14 Benefits – QUIC

- QUIC is more secure for its users.
- QUIC's connection set-up is faster.
- QUIC can evolve more easily.

Topic 6.15 HTTP Versions



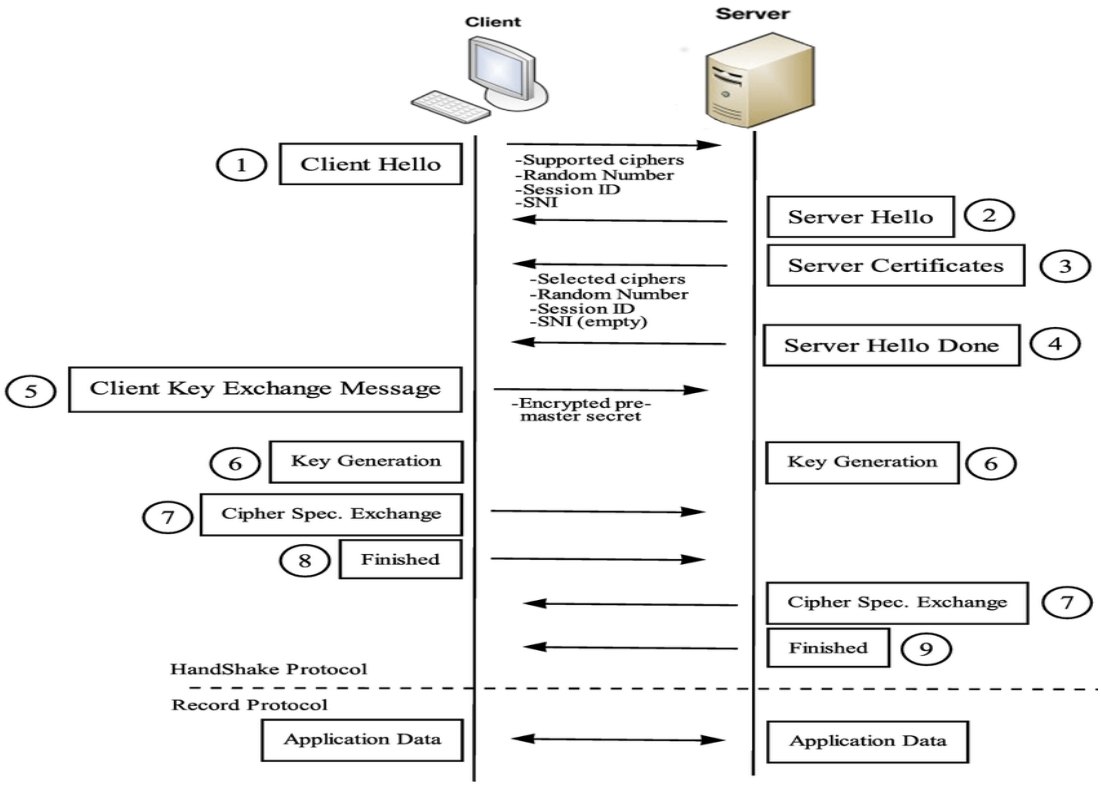
Topic 6.16 HTTPS

- Hypertext Transfer Protocol Secure is a secure version of HTTP.
- This protocol enables secure communication between a client (e.g. web browser) and a server (e.g. web server) by using encryption.
- HTTPS URLs begin with **https** instead of **http**.
- HTTPS uses a well-known TCP port 443.
- Make sure you always send requests over HTTPS and never ignore invalid certificates.
- HTTPS is the only thing protecting requests from being intercepted or modified!!

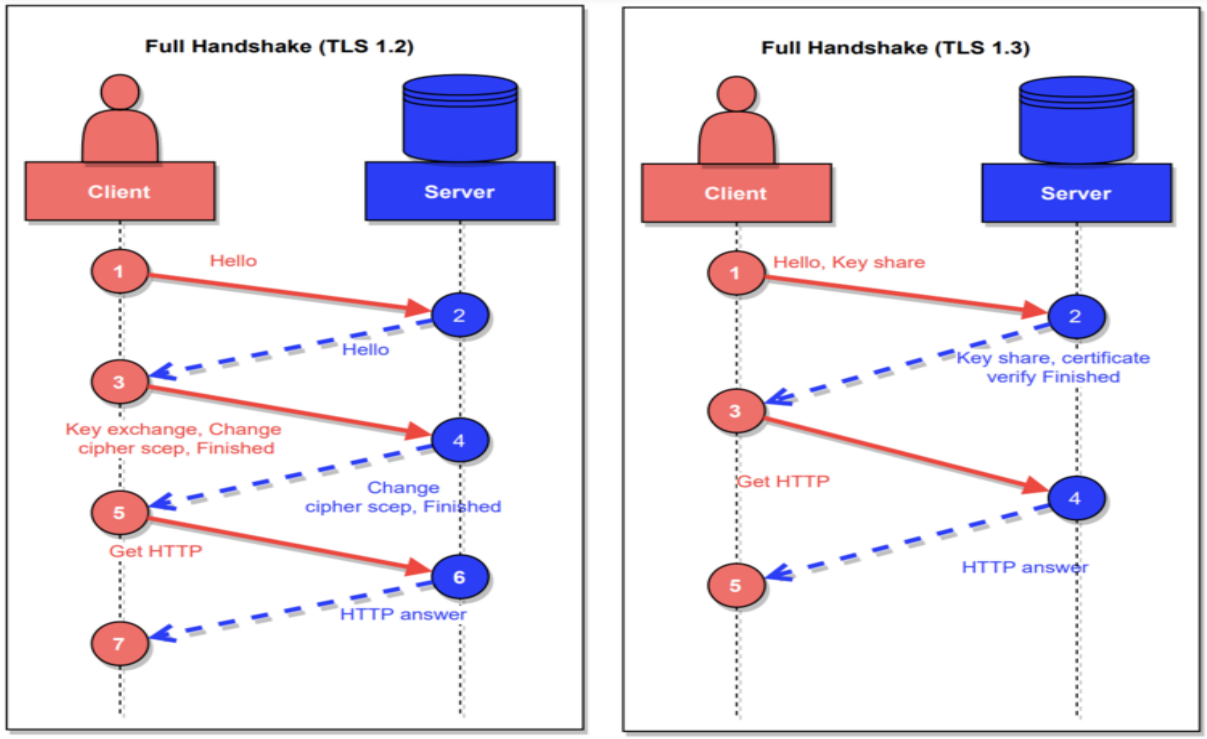
Topic 6.17 TLS handshake

- HTTPS uses **Transport Layer Security (TLS)** protocol or its predecessor **Secure Sockets Layer (SSL)** for encryption.
- The TLS handshake enables the TLS client and server to establish the secret keys with which they communicate.
- Agree on the version of the protocol to use.
- Select cryptographic algorithms.
- Authenticate each other by exchanging and validating digital certificates.
- Use asymmetric encryption techniques to generate a shared secret key, which avoids the key distribution problem.
- SSL or TLS then uses the shared key for the symmetric encryption of messages, which is faster than asymmetric encryption.

The TLS handshake



Topic 6.18 TLS Handshake



Topic 6.19 Certificates

- The HTTPS protocol is based on the server having a public key certificate, sometimes called an SSL certificate.
- When a secure connection is established, the server will send the client its TLS/SSL certificate.
- The client will then check the certificate against a trusted Certificate Authority, essentially validating the server's identity.
- A TLS/SSL certificate essentially binds an identity to a pair of keys which are then used by the server to encrypt as well as sign the data.
- A Certificate Authority is an entity that issues TLS/SSL or Digital certificates.

Topic 6.20 Summary

- HTTP connection Management
- HTTP/2
- QUIC
- HTTPS
- TLS Handshake

7. CS5 Web protocols

- HTTP
  - HTTP Request- Response and its structure
  - HTTP Methods;
  - HTTP Headers
  - Connection management - HTTP/1.1 and HTTP/2
- Synchronous and asynchronous communication
- Communication with Backend
  - AJAX, Fetch API
  - Webhooks
  - Server-Sent Events
- Polling
- Bidirectional communication - Web sockets

Topic 7.1 Calling API from Frontend

Topic 7.2 Fetching data from the server

- How do you call a service from a client?
- The browser makes one or more HTTP requests to the server for the files needed to display the page, and the server responds with the requested files.
- The browser reload the page with the new data.
- So, instead of the traditional model, many websites use JavaScript APIs to request data from the server and update the page content without a page load.
- **This is called AJAX**
- The Fetch API enables JavaScript on a page to request an HTTP server to retrieve specific resources.

Topic 7.3 Fetch API

- The Fetch API provides an interface for fetching resources
- Provides a generic definition of Request and Response objects
- The fetch() method takes one mandatory argument, the path to the resource you want to fetch.
- It returns a promise that resolves to the response of that request (successful or not).
- You can optionally pass an init options object as a second argument (used to configure req headers for other types of HTTP requests such as PUT, POST, DELETE)

Topic 7.4 Fetch API Example

```
fetch("https://www.boredapi.com/api/activity")
    .then(response => {
        return response.json();
    })
    .then(data => {
        console.log(JSON.stringify(data));
        document.getElementById("h1id").innerText = data.activity;
    }
    )
    .catch(error => {
        alert('There was a problem with the request.');
    });
```

Topic 7.5 Axios

- Axios is a promise-based HTTP client for JavaScript.
- It allows you to

• Make XMLHttpRequests from the browser

• Make http requests from node.js

• Supports the Promise API

• Automatic transforms for JSON data

- Axios provides more functions to make other network requests as well,

matching the HTTP verbs that you wish to execute, such as:

• axios.post(<uri>, <payload>)

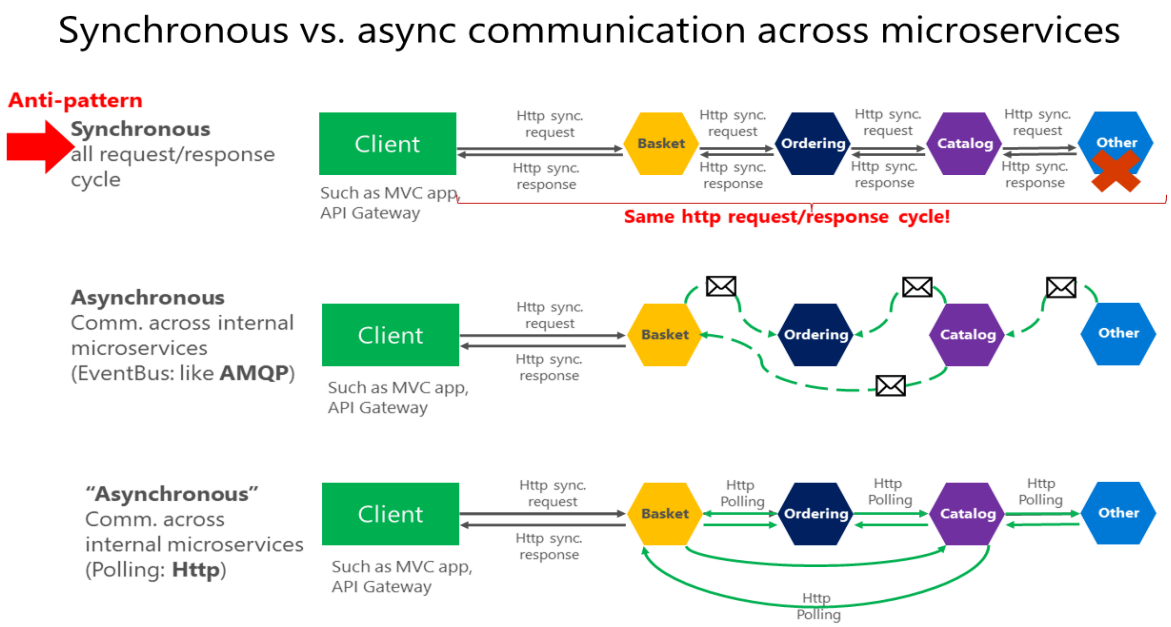
• axios.put(<uri>, <payload>)

• axios.delete(<uri>, <payload>)

Topic 7.6 Fetch vs Axios

- Fetch API is built into the window object, and therefore doesn't need to be installed as a dependency or imported in client-side code.
- Axios needs to be installed as a dependency.
- However, it automatically transforms JSON data.
- If you use .fetch() there is a two-step process when handing JSON data.
- The first is to make the actual request and then the second is to call the .json() method on the response.

Topic 7.7 Synchronous vs Asynchronous communication



Topic 7.8 Long running Transactions

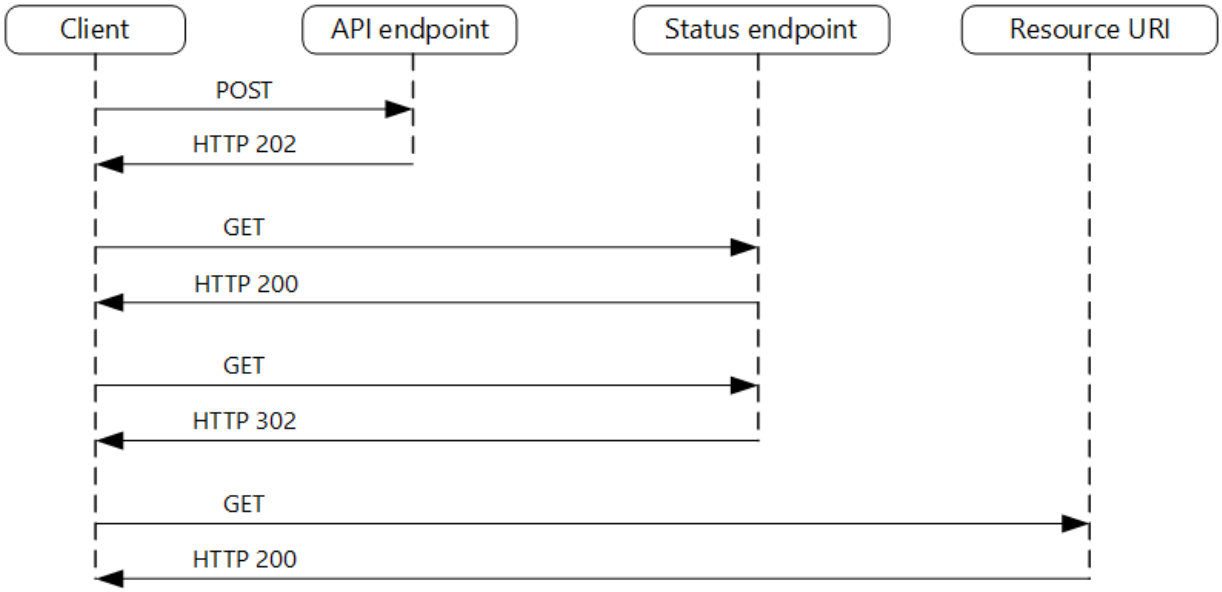
- API that performs tasks that could run longer than the request timeout limit.
- The server typically replies with a 202 Accepted Response indicating that the request is accepted and is in progress.
- HTTP is a **unidirectional** protocol, which means the client always initiates the communication.
- So client has to periodically check the server , if the work has been completed.

Topic 7.9 Approaches

- **How Server to Client Real Time Notification Response Works?**
- Polling
- Webhooks
- Server Sent Events
- Websockets
- Message Queues

Topic 7.10 HTTP Polling

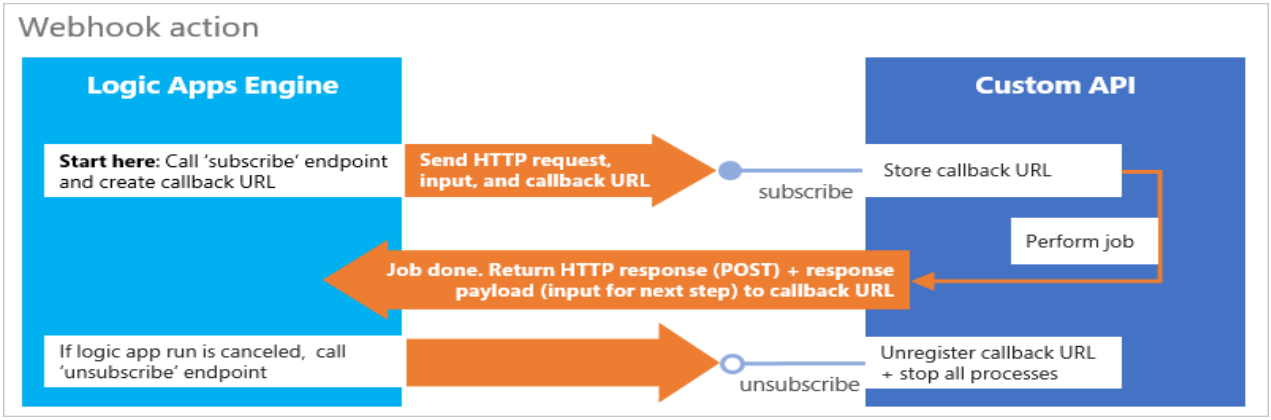
- HTTP Polling is a mechanism where the client requests the resource regularly at intervals.
- If the resource is available, the server sends the resource as part of the response.
- If the resource is not available, the server returns an empty response to the client.



- An HTTP 202 response should indicate the location and frequency that the client should poll for the response.
- It should have the following additional headers:
    - Location: A URL the client should poll for a response status.
    - Retry-After: This header is designed to prevent polling clients from overwhelming the back-end with retries.

Topic 7.11 WebHooks

- A webhook is an HTTP-based callback function that allows lightweight, event-driven communication between 2 application programming interfaces (APIs).
- This callback is an HTTP POST that sends a message to a URL when an event happens.
- To set up a webhook, the client gives a unique URL to the server API and specifies which event it wants to know about.
- Once the webhook is set up, the client no longer needs to poll the server;
- When the specified event occurs, the server will automatically send the relevant payload to the client's webhook URL.
- Webhooks are often called reverse APIs or push APIs because they put communication responsibility on the server rather than the client.



- Eliminate the need for polling
- Are quick to set up.
- Automate data transfer.

- Are good for lightweight, specific payloads.

- client-side” application is the one making the request to the API on the “server-side”.
- The “client-side” must be running a server, and the “server-side” must be running a server.
- The “client-side” application makes an API request to the “server-side” server, and sends the “server-side” server a “webhook” to call once the “server-side” wants to notify the “client-side” application of some “event”.
- Once the “event” occurs, and the “server-side” application calls the “webhook” url, the server that is running on the “client-side” application will “receive” that “webhook” notification.
- **Front end applications eg. pure React JS, AngularJS, Mobile Apps, cannot use webhooks directly.**
- Webhooks basically are APIs implemented by API consumers but defined and used by API providers to send notifications of events.

Topic 7.12 Server Sent Events

- Server-sent events (SSE) represent a unidirectional communication channel from the server to the client, allowing servers to push updates in real time.
- Unlike other technologies like WebSockets, SSE operates over a single HTTP connection.

Topic 7.13 Key Features of SSE

- **Simplicity**: Easy to implement and use.
- **Unidirectional Flow**: Data flows from server to client only.
- **Automatic Reconnection**: SSE connections automatically attempt to reconnect in case of interruptions.

Topic 7.14 Server-Sent Events

- Establishing the SSE Connection
- To initiate an SSE connection, the server sends a special text/event-stream MIME type response.
- On the client side, the EventSource API is used to handle incoming events.

Topic 7.15 Client Side

- To begin receiving events from the server , create a new EventSource object with the URL of a script that generates the events.
- Create an EventSource object to establish the SSE connection

```
const eventSource = new EventSource('http://localhost:3000');
```

- To receive message events, attach a handler for the message event:

```
eventSource.onmessage = (event) => {
  const data = JSON.parse(event.data);
document.getElementById('output').innerHTML = `Received data: ${data.message}`;
};
```

- This code listens for incoming message events and appends the message text to a list in the document's HTML.

Topic 7.16 Server Side

- The server-side script that sends events must respond using the MIME type text/event-stream.
- Each notification is sent as a block of text terminated by a pair of newlines.
- Event stream format
- data: {"message":"Server time: Thu Jan 18 2024 00:37:43 GMT+0530 (India Standard Time)"}
- A pair of newline characters separate messages in the event stream.

Topic 7.17 Use Cases and Applications

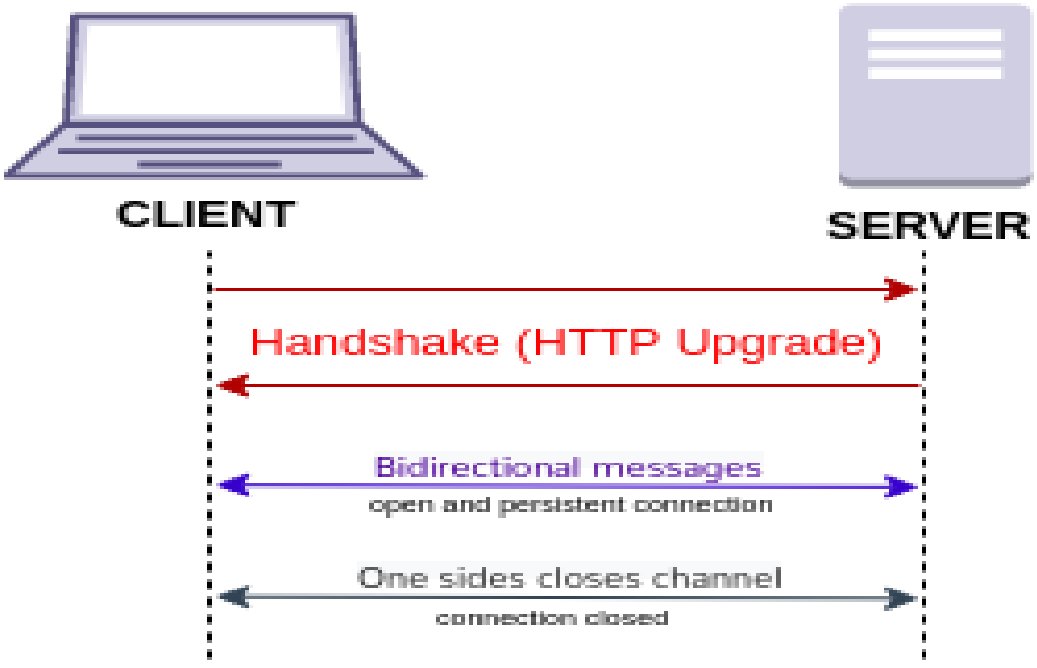
- Real-Time Notifications
- Live Feeds and Dashboards
- Collaborative Editing and Chat Applications

Topic 7.18 WebSockets

- WebSocket is a communications protocol that supports bidirectional communication over a single TCP connection.
- It is a living standard maintained by the WHATWG and a successor to The WebSocket API from the W3C.
- The WebSocket protocol enables full-duplex interaction between a web browser (or other client application) and a web server

Topic 7.19 WebSockets Handshake

- The WebSocket protocol specification defines ws (WebSocket) and wss (WebSocket Secure) as two new uniform resource identifier (URI) schemes.
- ws : [// authority] path [? query]



Topic 7.20 WebSocket – Upgrade

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

Origin: http://example.com

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

Sec-WebSocket-Protocol: chat

Topic 7.21 Headers

- The Sec-WebSocket-Key header is calculated by base64 encoding a random string of 16 characters, each with an ASCII value between 32 and 127.
- A different key must be randomly generated for each connection.
- The Sec-Websocket-Accept header is included with the initial response from the server.
- The server reads the Sec-WebSocket-Key, appends UUID 258EAFA5-E914-47DA-95CA- to it, re-encodes it using base64, and returns it in the response as the parameter for Sec-Websocket-Accept
- The client sends the Sec-WebSocket-Protocol header to ask the server to use a specific subprotocol.

Topic 7.22 Summary

- HTTP Polling
- Webhooks
- Server-Sent Events
- Websockets

8. CS6 REST – Concepts

- REST stands for REpresentational State Transfer
- REST is an architectural style.
- A RESTful web service
  - ✓ exposes information about itself in the form of information about its resources
  - ✓ enables the client to take actions or perform CRUD operations on those resources.
  - ✓ Resource
  - ✓ Can be anything(docs, data, media, images) the web service can provide information about
  - ✓ Each resource has a unique identifier - can be a name or a number

When a RESTful API is called, the server will transfer to the client a representation of the state of the requested resource!

Topic 8.1 REST – example

- When a developer calls Instagram API to fetch a specific user (the resource)
  - ✓ the API will return the state of that user, including
    - their name
    - the number of posts that user posted on Instagram so far
    - how many followers they have
    - and more
  - ✓ The representation of the state can be in a JSON format
  - ✓ probably for most APIs this is indeed the case
  - ✓ but can also be in XML or HTML format

Topic 8.2 Resources

- The main building blocks of a REST architecture are the resources
- **Representations:** It can be any way of representing data (binary, JSON, XML, etc.).
- A single resource can have multiple representations.
- **Identifier:** A URL that retrieves only one specific resource at any given time.
- **Metadata:** Content type, last-modified time, and so forth.
- **Control data:** Is-modifiable-since, cache-control.

Topic 8.3 REST and HTTP

- The fundamental principle of REST is to use the **HTTP** protocol for data communication.
- RESTful web service makes use of HTTP for determining the action to be carried out on the particular resources
- REST gets its motivation from HTTP. Therefore, it can be said as a structural pillar of the REST
- Commonly Used ones
  - ✓ GET - to read (or retrieve) a representation of a resource
  - ✓ POST - utilized to create new resources
  - ✓ PUT - to update a resource
  - ✓ PATCH - to modify a resource - only needs to contain the changes to the resource, not the complete resource

- ✓ DELETE - to delete a resource identified by a URI.

- Rarely Used ones
  - ✓ Head – requests the headers.
  - ✓ Options - requests permitted communication options for a given URL or server

Topic 8.4 REST

- Server responds based on the identifier and the operation
  - ✓ An identifier for the resource you are interested in
    - URL for the resource, also known as the endpoint
    - URL stands for Uniform Resource Locator
    - A REST service exposes a URI for every piece of data the client might want to operate on.(Scoping Information)
  - ✓ The operation you want the server to perform on that resource
    - in the form of an HTTP method, or verb
    - common HTTP methods are GET, POST, PUT, and DELETE
    - One way to convey method information in a web service is to put it in the HTTP method(Method Information)

Topic 8.5 Method Information

- In a SOAP, REST, RPC
- How the client can convey its intentions to the server?

Topic 8.6 REST Get Request

An HTTP GET request for http://www.oreilly.com/index.html

GET /index.html HTTP/1.1

Host: www.oreilly.com

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.12)...

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,...

Accept-Language: us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive

The response to an HTTP GET request for http://www.oreilly.com/index.html

HTTP/1.1 200 OK

Date: Fri, 17 Nov 2006 15:36:32 GMT

Server: Apache

Last-Modified: Fri, 17 Nov 2006 09:05:32 GMT

Etag: "7359b7-a7fa-455d8264

Accept-Ranges: bytes

Content-Length: 43302

Content-Type: text/html

X-Cache: MISS from www.oreilly.com

Keep-Alive: timeout=15, max=1000

Connection: Keep-Alive

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>

...

<title>oreilly.com -- Welcome to O'Reilly Media, Inc.</title>

Topic 8.7 SOAP Request

A sample SOAP RPC call

POST search/beta2 HTTP/1.1

Host: api.google.com

Content-Type: application/soap+xml

SOAPAction: urn:GoogleSearchAction

<?xml version="1.0" encoding="UTF-8"?>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<gs:doGoogleSearch xmlns:gs="urn:GoogleSearch">

<q>REST</q>

...

</gs:doGoogleSearch>

</soap:Body>

</soap:Envelope>

Part of the WSDL description for Google's search service

<operation name="doGoogleSearch">

<input message="typens:doGoogleSearch"/>

<output message="typens:doGoogleSearchResponse"/>

</operation>

Topic 8.8 XML-RPC Example

POST /rpc HTTP/1.1

Host: www.upcdatabase.com

User-Agent: XMLRPC::Client (Ruby 1.8.4)

Content-Type: text/xml; charset=utf-8

Content-Length: 158

Connection: keep-alive

<?xml version="1.0" ?>

<methodCall>

<methodName>lookupUPC</methodName>

...

</methodCall>

Example 1-12. An XML document describing an XML-RPC request

<?xml version="1.0" ?>

<methodCall>

<methodName>lookupUPC</methodName>

<params>

<param><value><string>001441000055</string></value></param>

</params>

</methodCall>

Topic 8.9 Scoping Information

- *http://flickr.com/photos/tags/penguin*
- *http://api.flickr.com/services/rest/?method=flickr.photos.search&tags=penguin*
- http://www.upcdatabase.com/upc/00598491'
- One obvious place to put it is in the URI path.
- A RESTful, resource-oriented service exposes a URI for every piece of data the client might want to operate on.

Topic 8.10 Method and Scoping information

- In RESTful web service,
  - ✓ the method information goes into the HTTP method.
  - ✓ The scoping information goes into the URI.
- Given the first line of an HTTP request to a RESTful web service you should understand basically what the client wants to do.
- "GET /reports/docs HTTP/1.1"
- If the HTTP method doesn't match the method information, the service isn't RESTful.
- The service is not resource-oriented if the scoping information isn't in the URI.

Topic 8.11 Key principles

- Everything is a resource
- Each resource is identifiable by a unique identifier (URI)
- Use the standard HTTP methods

- Resources can have multiple representations
- Communicate statelessly

Topic 8.12 REST Constraints

- For a web service to be RESTful, it has to adhere to 6 constraints:
  - ✓ Client - Server separation
  - ✓ Stateless
  - ✓ Cacheable
  - ✓ Uniform interface
  - ✓ Layered system
  - ✓ Code-on-demand (Optional)

Topic 8.13 Client - Server separation

- The client and the server act independently, each on its own
  - ✓ Interaction between them is only in the form of
    - requests initiated by the client only
    - responses, which the server send to the client only as a reaction to a request
- The server sits there waiting for requests from the client to come
  - ✓ doesn't start sending away information about the state of some resources on its own
  - ✓ Responds only when a request comes in

Topic 8.14 Stateless

- Stateless means the server does not remember anything about the user who uses the service
  - ✓ doesn't remember if the user already sent a GET request for the same resource in the past
  - ✓ doesn't remember which resources the user of the API requested before
  - ✓ and so on...
- Each individual request contains all the information the server needs to perform the request and return a response, regardless of other requests made by the same user.
- The client is responsible for sending any state information to the server whenever it's needed.
- No session stickiness or session affinity on the server for the calling request

Topic 8.15 Cacheable

- Data the server sends contain information about whether or not the data is cacheable
- If the data is cacheable, it might contain some version number
  - ✓ version number is what makes caching possible
- The client knows which version of the data it already has (from a previous response)
  - ✓ the client can avoid requesting the same data again and again
- client should also know if the current version of the data is expired,

- ✓ will know it should send another request to the server to get the most updated data about the state of a resource

Topic 8.16 Uniform interface

- There are four guiding principles suggested by Fielding that constitute the necessary constraints to satisfy the uniform interface
  - Identification of resources
  - Manipulation of resources
  - Self-descriptive messages
  - Hypermedia as the engine of application state
- The request to the server has to include a resource identifier
- Each request to the web service contains all the information the server needs to perform the request
  - Each response the server returns contain all the information the client needs to understand the response
- The response the server returns include enough information so the client can manipulate the resource
- Hypermedia as the engine of application state
  - Application mean the web application that the server is running
  - Hypermedia mean the hyperlinks, or simply links, that the server can include in the response
  - means that the server can inform the client , in a response, of the ways to change the state of the web application

Topic 8.17 HATEOS

- Hypermedia as the Engine of Application State

Without HATEOAS:

```
Request:
[Headers]
user: jim
roles: USER
GET: /items/1234
Response:
HTTP 1.1 200
{
    “id” : 1234,
    “description” : “FooBar TV”,
    “image” : “fooBarTv.jpg”,
    “price” : 50.00,
    “owner” : “jim”
}
```

With HATEOAS:

```
Request:
[Headers]
user: jim
roles: USER
GET: /items/1234
Response:
HTTP 1.1 200
{
    “id” : 1234,
    “description” : “FooBar TV”,
    “image” : “fooBarTv.jpg”,
    “price” : 50.00,
    “links” : [
            {
                “rel” : “modify”,
                “href” : “/items/1234”
            },
            {
                “rel” : “delete”,
                “href” : “/items/1234”
            }
        ]
    }
}
```

Topic 8.18 Layered system

- Between the client who requests a representation of a resource’s state, and the server who sends the response back
  - ✓ there might be a number of servers in the middle
  - ✓ servers might provide a security layer, a caching layer, a load-balancing layer etc.,
  - ✓ layers should not affect the request or the response

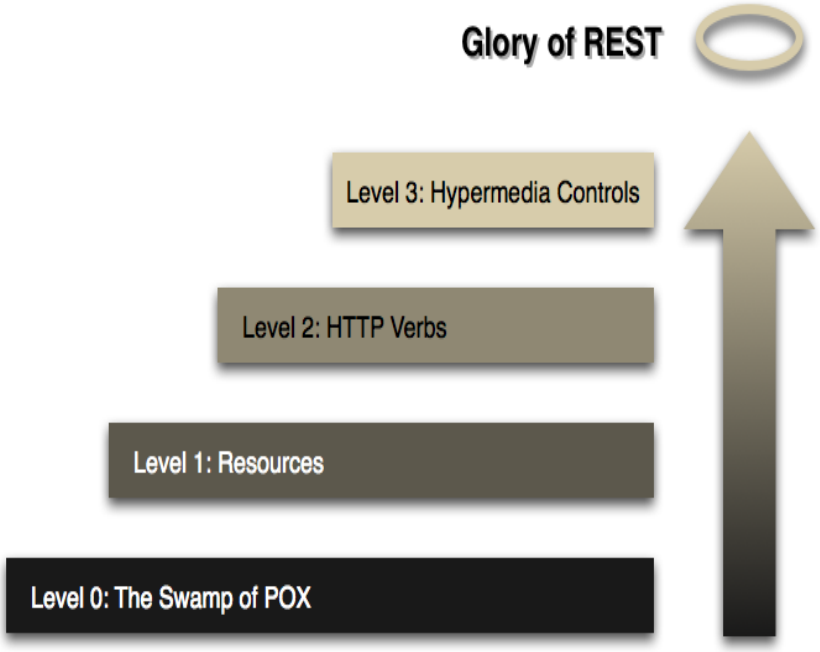
- The client is agnostic as to how many layers, if any, there are between the client and the actual server responding to the request.

Topic 8.19 Code-on-demand (Optional)

- Is optional — a web service can be RESTful even without providing code on demand
- The client can request code from the server
  - ✓ response from the server will contain some code
  - ✓ when the response is in HTML format, usually in the form of a script
- The client then can execute that code

Topic 8.20 REST Maturity Model

- Richardson used three main factors to decide the maturity of a service. These factors are
- URI,
- HTTP Methods,
- HATEOAS (Hypermedia)



Topic 8.21 Summary

REST Constraints

REST Principles

9. CS7 REST Design

Topic 9.1 Agenda

- Interaction Design with HTTP(response codes, request methods)
- Identifier Design with URIs
- Metadata Design(Media Types, content negotiation)
- Representation Design(Message body format, Hypermedia Representation)

Topic 9.2 REST API Design

- API design is more of an art.
- Some best practices for REST API design are implicit in the HTTP standard.

• When should URI path segments be named with plural nouns?

• Which request method should be used to update the resource state?

• How do I map *non-CRUD* operations to my URIs?

• What is the appropriate HTTP response status code for a given scenario?

• How can I manage the versions of a resource's state representations?

• How should I structure a hyperlink in JSON?

Topic 9.3 API Design

- REST APIs are designed around *resources* , which is any kind of object, data, or service that is accessible to the client.
- A resource has an *identifier* , a URI that uniquely identifies that resource.
- The resource URIs should be based on nouns (the resource) and not verbs (the operations on the resource).

Topic 9.4 API Design Elements

- The following aspects of API design are all important, and together they define your API:
    - The representations of your resources and the links to related resources.
    - The use of standard (and occasionally custom) HTTP headers.
    - The URLs and URI templates define your API's query interface for locating resources based on their data.
    - Required behaviors by clients—for example, DNS caching behaviors, retry behaviors

- Interaction Design with HTTP(response codes, request methods)
- Identifier Design with URIs
- Metadata Design(Media Types, content negotiation)
- Representation Design(Message body format, Hypermedia Representation)

Topic 9.5 Interaction Design

- Interaction Design with HTTP
- REST APIs embrace all aspects of the HyperText Transfer Protocol including its request methods, response codes, and message headers
- Each HTTP method has specific, well-defined semantics within the context of a REST API's resource model.
- The purpose of GET is to retrieve a representation of a resource's state.

- HEAD is used to retrieve the metadata associated with the resource's state.
- PUT should be used to add a new resource to a store or update a resource.
- DELETE removes a resource from its parent.
- POST should be used to create a new resource within a collection and execute controllers.

- Interaction Design with HTTP
- Rule: GET and POST must not be used to tunnel other request methods
- Rule: GET must be used to retrieve a representation of a resource
- Rule: HEAD should be used to retrieve response headers
- Rule: PUT must be used to both insert a stored resource
- Rule: POST must be used to create a new resource in a collection
- Rule: POST must be used to execute controllers
- Rule: DELETE must be used to remove a resource from its parent
- Rule: OPTIONS should be used to retrieve metadata that describes a resource's available interactions

Topic 9.6 HTTP response success code summary

- 200 OK Indicates a nonspecific success
- 201 Created Sent primarily by collections and stores but sometimes also by controllers to indicate that a new resource has been created
- 204 No Content Indicates that the body has been intentionally left blank
- 301 Moved Permanently Indicates that a new *permanent* URI has been assigned to the client's requested resource
- 303 See Other Sent by controllers to return results that it considers optional
- 401 Unauthorized Sent when the client either provided invalid credentials or forgot to send them
- 402 Forbidden Sent to deny access to a protected resource
- 404 Not Found Sent when the client tried to interact with a URI that the REST API could not map to a resource
- 405 Method Not Allowed Sent when the client tried to interact using an unsupported HTTP method
- 406 Not Acceptable Sent when the client tried to request data in an unsupported media type format
- 500 Internal Server Error Tells the client that the API is having problems of its own

Topic 9.7 Identifier Design

- REST APIs use Uniform Resource Identifiers (URIs) to address resources
- URI Format
- URI = scheme "://" authority "/" path ["?" query] ["#" fragment]
- Forward slash separator (/) must be used to indicate a hierarchical relationship.
- Consistent subdomain names should be used for your APIs

Topic 9.8 Resource Modelling

The URI path conveys a REST API's resource model,

Each forward slash-separated path segment corresponds to a unique resource within the model's hierarchy.

http://api.library.restapi.org/books/harry-potter

http://api.library.restapi.org/books

http://api.library.restapi.org/sections/fiction

http://api.library.restapi.org/sections

document,

http://api.library.restapi.org/books/Pride and Prejudice

http://api.library.restapi.org/books/harry-potter/harry potter and the philosopher's stone

collection,

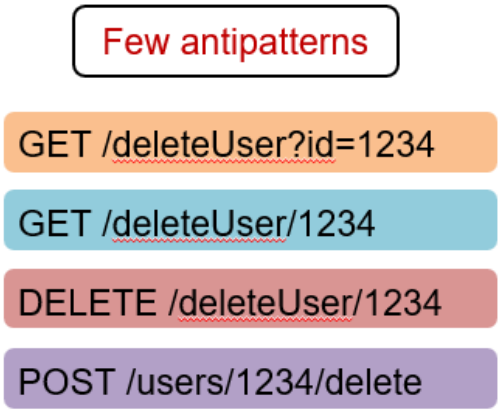
Each URI below identifies a collection resource:

http://api.library.restapi.org/books

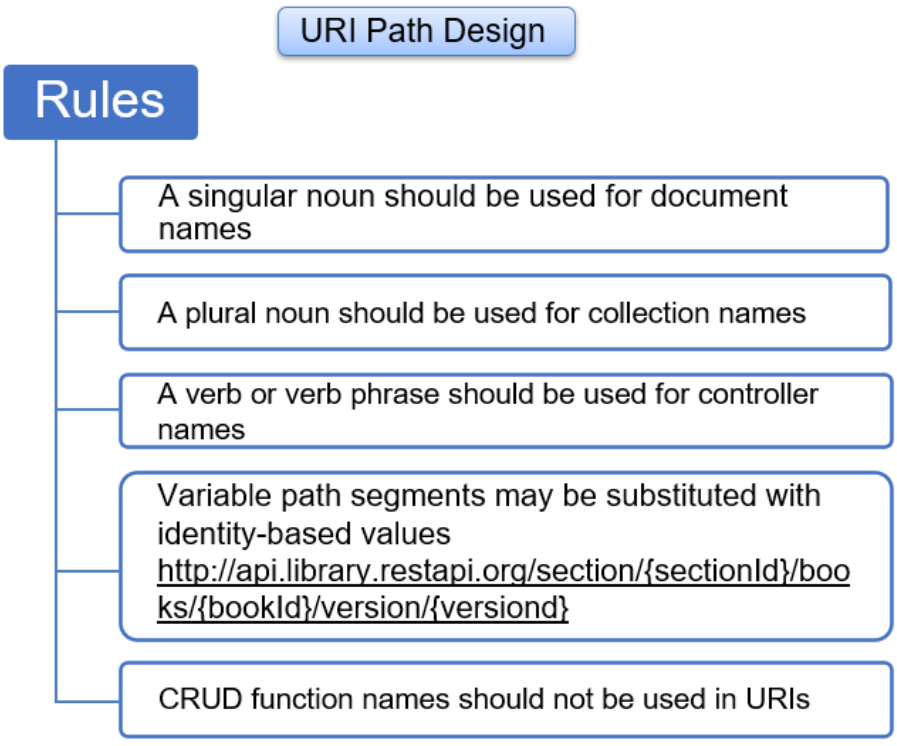
http://api.library.restapi.org/sections

controller.

POST /alerts/245743/resend



Topic 9.9 Identifier Design



Topic 9.10 Identifier Design

- URI Query Design
- As a component of a URI, the query contributes to the unique identification of a resource.
- Rule: The query component of a URI may be used to filter collections
- Rule: The query component of a URI should be used to paginate collections
- Represent relationships clearly in the URI

Topic 9.11 Filter and Paginate

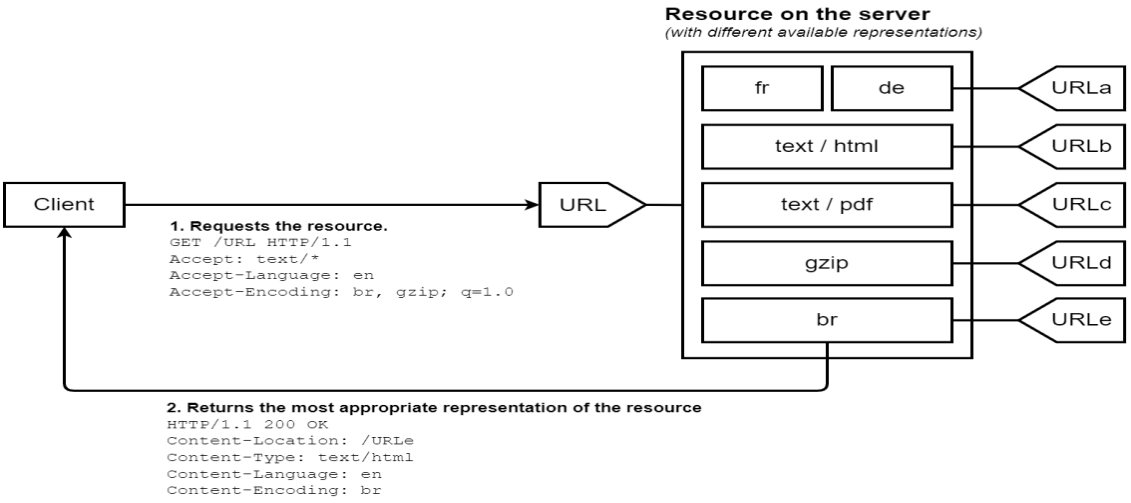
- */orders?minCost=n*
- /orders?limit=25&offset=50

Topic 9.12 Metadata Design

- Important HTTP headers
    - Content type
    - Content length
    - Last-modified
    - E-tag
    - Location

Topic 9.13 Metadata Design

- Media type negotiation should be supported when multiple representations are available



Topic 9.14 Representation Design

- Representation is the technical term for the data that is returned
- Users can view the representation of a resource in different formats, called media types
- JSON is the preferred format
- XML and other formats may optionally be used for resource representation
- A consistent form should be used to represent media types, links, and errors

Topic 9.15 Representation Design

- Keep the choice of media types and formats flexible to allow for varying application use cases, and client needs for each resource.
- Prefer to use well-known media types for representations.
- What data to include in JSON-formatted representations

- An example of a representation of a person resource:

{

"name" : "John",

"id" : "urn:example:user:1234",

"link" : {

"rel : "self",

"href" : "http://www.example.org/person/john"

},

"address" : {

"id" : "urn:example:address:4567",

"link" : {

"rel : "self",

"href" : "http://www.example.org/person/john/address"

}

Topic 9.16 Best Practices – Summary

- Use only nouns for a URI;
- GET methods should not alter the state of resource;
- Use plural nouns for a URI;
- Use sub-resources for relationships between resources;
- Use HTTP headers to specify input/output format;
- Provide users with filtering and paging for collections;
- Version the API;
- Provide proper HTTP status codes.

Topic 9.17 Best Practices

- Consistent API s - reduces the cognitive load on developers.
- Meaningful errors

Example:

Authentication failed because token is revoked : token_revoked or invalid_auth

Value passed for name exceeded max: length name_too_long or invalid_name

Topic 9.18 API Design

- "*The API should enable developers to do one thing really well.It's not as easy as it sounds, and you want to be clear on what the API is not going to do as well."*
- *- Ido Green, developer advocate at Google*

Topic 9.19 RESTful Services Example – Customer

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

10. CS8 API Documentation

Topic 10.1: Agenda

- REST Design
- Documentation
- Versioning

Topic 10.2: REST Design

- Both object-oriented design and database modeling techniques use domain entities as a basis for design.
- You can use the same technique to identify resources.
- Analyze your use cases to find domain nouns that can be operated using CRUD operations
- Discussion

Consider a web service for managing photos. Clients can upload a new photo, replace

an existing photo, view a photo, or delete a photo.

In this example, "photo" is an entity in the application domain.

The actions a client can perform on this entity include "create a new photo," "replace an existing photo," "view a photo," and "delete a photo."

Topic 10.3: REST Design

- *Provide a way for a client to get a user's profile with a minimal set of properties.*
- *List of the 10 latest photos uploaded by the user.*
- In all these use cases, it is easy to spot the nouns.
- But in each case you will find that the corresponding actions do not map to the basic CRUD operations
- Bluntly mapping domain entities into resources may lead to resources that are inefficient and inconvenient to use.
- You will need additional resources to tackle such use cases.
- A resource doesn't have to be based on a single physical data item.
- For example, an order resource might be implemented internally as several tables in a relational database, but presented to the client as a single entity.
- Avoid creating APIs that simply mirror the internal structure of a database.
- The purpose of REST is to model entities and the operations that an application can perform on those entities.
- Designing Collections
- A collection is a separate resource from the item within the collection, and should have its own URI.
- The relationships between different types of resources
- /customers/1/orders/99/products --- Cumbersome!!

- /customers/1/orders to find all the orders for customer 1
- /orders/99/products to find the products in this order.
- **Use HATEOAS to enable navigation to related resources**
- *collection/item/collection* is suitable, avoid more complexity

Topic 10.4: API Versioning

- Breaking changes primarily fit into the following categories:
    - Changing the request/response format (e.g. from XML to JSON)
    - Changing a property name (e.g. from name to productName) or data type on a property (e.g. from an integer to a float)
    - Adding a required field on the request (e.g. a new required header or property in a request body)
    - Removing a property on the response (e.g. removing description from a product)

Topic 10.5: API Change Management

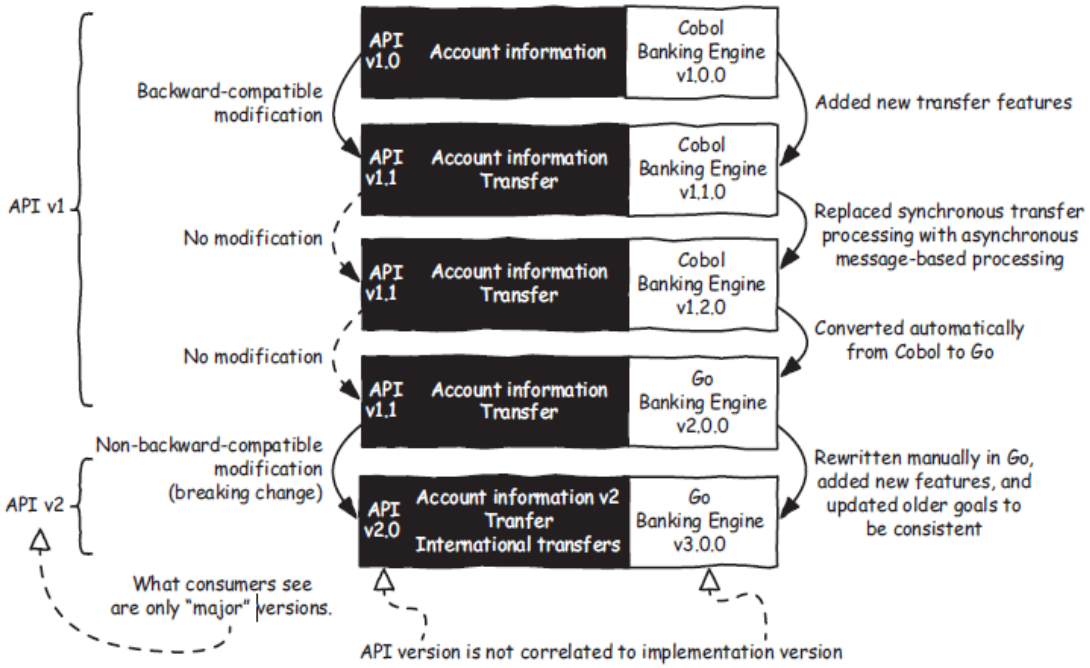
- Effective change management in the context of an API is summarized by the following principles:
- Continue support for existing properties/endpoints
- Add new properties/endpoints rather than changing existing ones
- Thoughtfully sunset obsolete properties/endpoints
- We can think of levels of scope change within a tree analogy:
- Leaf - A change to an isolated endpoint with no relationship to other endpoints
- Branch - A change to a group of endpoints or a resource accessed through several endpoints
- Trunk - An application-level change, warranting a version change on most or all endpoints
- Root - A change affecting access to all API resources of all versions
- As you can see, moving from leaf to root, the changes become progressively more impactful and global in scope.

Topic 10.6: API versioning representation

- There are four different ways that we can implement the versioning
- **Versioning through the URI path**
    - http://www.example.org/v1/customer/1234
    - Ideal when major changes are introduced that completely break backward compatibility.
- **Versioning through query parameters**
    - http://www.example.org/customer/1234?version=v3
    - Ideal for simple APIs where backward compatibility is maintained across versions.
- **Versioning through custom headers**
    - Custom-Header: api-version=1
- **Versioning through content-negotiation**

- Accept: application/vnd.adventure-works.v1+json

Topic 10.7: API Versioning



Topic 10.8: Semantic Versioning

- format: *MAJOR.MINOR.PATCH*.
- The MAJOR digit is incremented only on breaking changes, such as adding a new mandatory parameter
- The MINOR digit is incremented when new features are added in a backward-compatible manner, like adding new HTTP methods or resource paths in a REST API.
- The PATCH digit is incremented when the modifications made involve backward-compatible bug
- This makes sense for an implementation, but not for an API.
- Semantic versioning applied to APIs consist of just two digits: *BREAKING.NONBREAKING*.
- This two-level versioning is interesting from the provider's perspective; it helps to keep track of all the different backward-compatible and non-backward-compatible versions of an API.

Topic 10.9: API Documentation

- API documentation is a technical content deliverable containing instructions on using and integrating with an API effectively.
- API description formats like the OpenAPI/Swagger Specification have automated the documentation process, making it easier for teams to generate and maintain them.
- OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs.

Topic 10.10: OAS

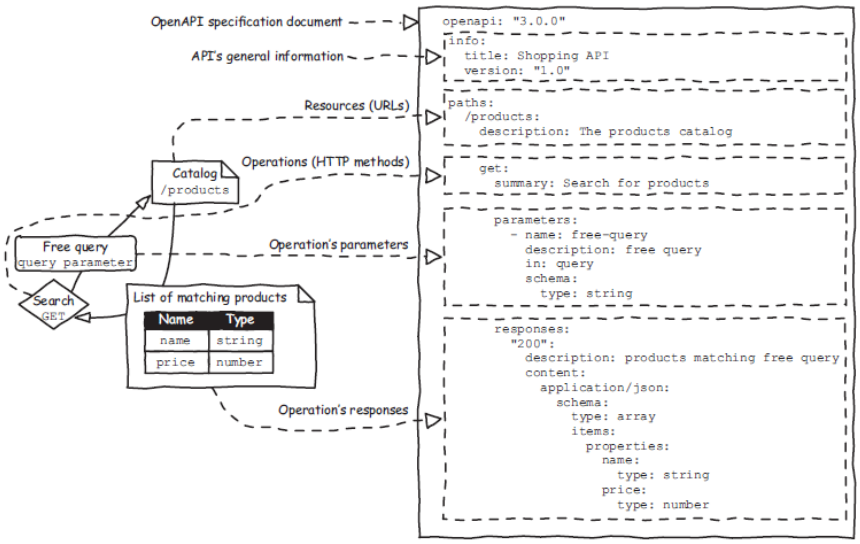
- The *OpenAPI Specication* (OAS) is a popular REST API description format
- An OpenAPI file allows you to describe your entire API, including:
    - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
    - Operation parameters Input and output for each operation
    - Authentication methods
    - Contact information, license, terms of use and other information.
    - API specifications can be written in YAML or JSON.

Topic 10.11: Swagger

- **Swagger** is a set of open-source tools built around the OpenAPI Specification that can help you design, build, document and consume REST APIs.
- http://swagger.io/docs/specification/basic-structure/

Topic 10.12: API Documentation

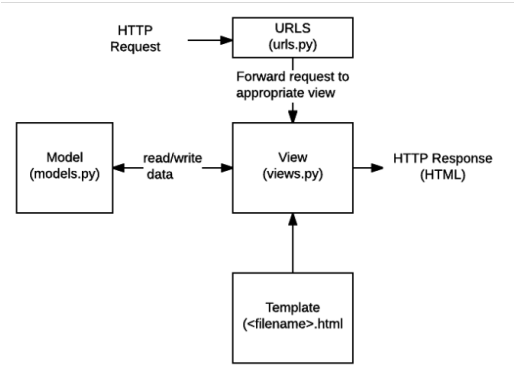
- An OAS document describing the search for products goal of the Shopping API



Topic 10.13: API Documentation

- RAML and Blueprint are other alternatives
- API Blueprint is a markdown format to generate documentation. It was developed by Apiary in 2013 and then acquired by Oracle.
- RAML, which stands for RESTful API Modeling Language, is an API design format developed by MuleSoft in 2013 and then acquired by Salesforce.
- Swagger is an API description format developed by Wordnik in 2010 and then acquired by SmartBear. It was later renamed OpenAPI and donated to the Linux Foundation.

Topic 10.14: Django Application



11. CS9 gPRC

Topic 11.1: Agenda

gRPC

Topic 11.2: gRPC- An RPC library and framework

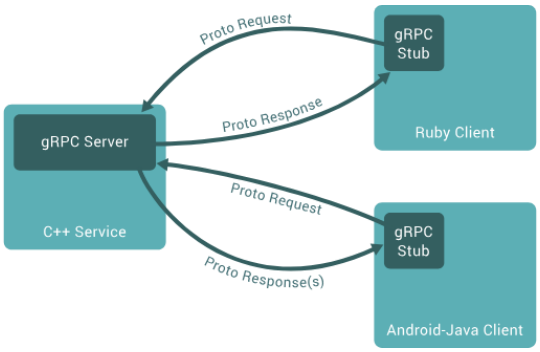
- Problems with RPC- laid the motivation for gRPC
- In 2015, Google released gRPC , a modern, open source, high-performance remote procedure call (RPC) framework that can run anywhere.
- An interprocess communication technology that allows you to connect, invoke, operate, and debug distributed heterogeneous applications as easily as making a local function call.

Topic 11.3: gRPC

- What does the "g" in gRPC actually stand for?
    - Its not Google
    - Google changes the meaning of the "g" for each version
    - Refer to the README doc

Topic 11.4: gRPC- Architecture

- A service Definition is created.
- Using that service definition, the server-side code known as a *server skeleton* is generated
- On the server side, the server implements the service, and runs a gRPC server to handle client calls.
- On the client side, the client has a stub that provides the same methods as the server.
- gRPC uses **Protocol Buffers** as Interface Definition Language (**IDL**) and message interchange format.



Topic 11.5: Protocol Buffers

- gRPC uses protocol buffers as the Interface Definition Language to define the service interface
- Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.
- The service interface definition is specified in a proto file -an ordinary text file with a .proto extension.

- The service definition specified in the .proto file, is used by both the server and client sides to generate the code.
- Protocol buffer data is structured as messages.
- Each message is a small logical record of information containing a series of name-value pairs called fields.
- Example:

```
message Person {
  string name = 1;
  int id = 2;
  string description = 3;
}
```

- You define gRPC services in ordinary proto files, with RPC method parameters and return types specified as protocol buffer messages.

Example:

```
service ProductInfo {
rpc addProduct(Product) returns (ProductID);
rpc getProduct(ProductID) returns (Product);
}
message Product {
string id = 1;
string name = 2;
string description = 3;
}
message ProductID {
string value = 1;
}
```

Topic 11.6: gRPC Service

- The service definition can be used to generate the server or client-side code using the protocol buffer compiler protoc.
- Since gRPC service definitions are language agnostic, you can generate clients and servers for any supported language
- Server Side:
    - Implement the service logic of the generated service skeleton by overriding the service base class.
    - Run a gRPC server to listen for requests from clients and return the service responses.
- Client Side:

- The client stub provides the same methods as the server, which your client code can invoke.
- The client stub translates them to remote function invocation network calls that go to the server side.

Topic 11.7: Why gRPC?

- Efficient for inter-process communication
  - gRPC implements protocol buffers on top of HTTP/2
- Simple, well-defined service interfaces and schema
  - contract-first approach
- Strongly typed- protocol buffers to define gRPC services
- Polyglot
- Duplex streaming
- Built-in commodity features

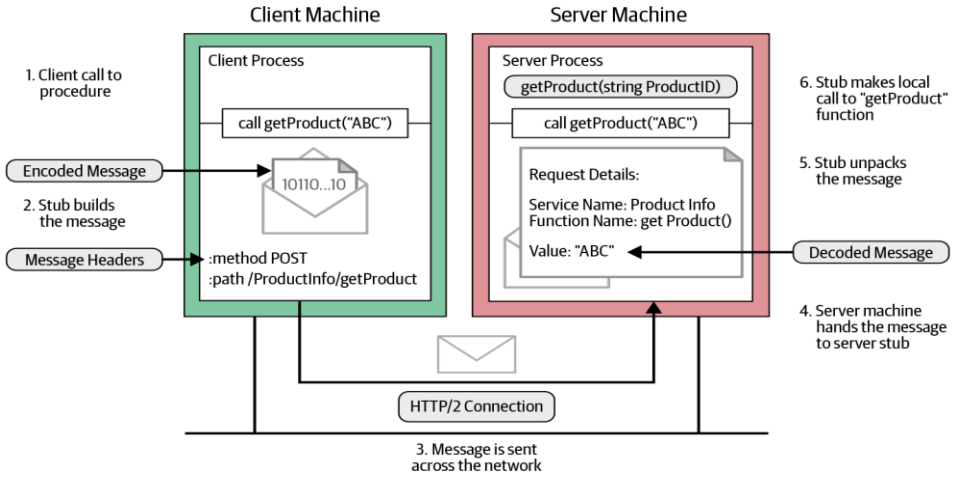
Topic 11.8: gRPC in Real World

- With the adoption of gRPC, Netflix has seen a massive boost in developer productivity.
- Etcd uses a gRPC user-facing API to leverage the full power of gRPC.
- Dropbox has switched to gRPC

Topic 11.9: gRPC- Disadvantages

- It may not be suitable for external-facing services.
- Drastic service definition changes are a complicated development process
- The ecosystem is still evolving.

Topic 11.10: gRPC- Flow



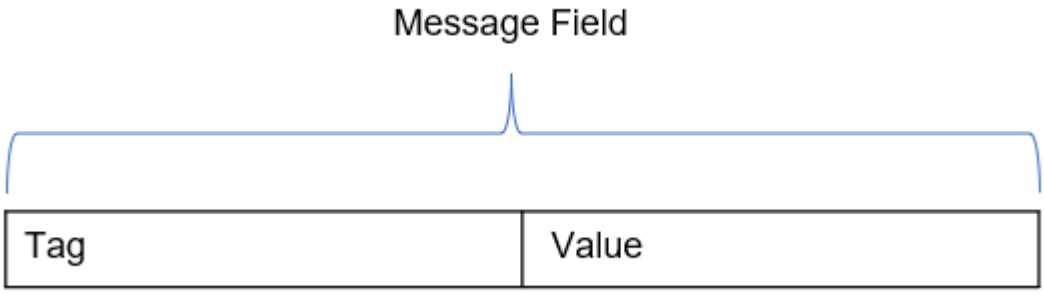
Message Encoding

message ProductID {

string value = 1;

}

Let’s say we need to get product details for product ID 15;



Topic 11.11: Message Encoding

- Tag value = (field_index << 3) | wire_type

- **Wire type**
  - 0 for Varint int32, int64, uint32, uint64, sint32, sint64, bool, enum
  - 1 for 64-bit fixed64, sfixed64, double
  - 2 for Length-delimited string, bytes, embedded messages, packed repeated fields
  - 3 for Start group groups (deprecated)
  - 4 for End group groups (deprecated)
  - 5 for 32-bit fixed32, sfixed32, float
  - Tag value = (00000001 << 3) | 00000010
- = 000 1010

Topic 11.12: Length-Prefixed Message Framing

- gRPC uses a message-framing technique called length-prefix framing.
- Length-prefix is a message-framing approach that writes the size of each message before writing the message itself.

Topic 11.13: gRPC over HTTP/2

- gRPC uses HTTP/2 as its transport protocol to send messages over the network.
- This is one of the reasons why gRPC is a high-performance RPC framework.
- The gRPC channel represents a connection to an endpoint, which is an HTTP/2 connection.
- Messages that are sent in the remote call are sent as HTTP/2 frames.
- A frame may carry one gRPC length-prefixed message, or if a gRPC message is quite large it might span multiple data frames.

Topic 11.14: Communication patterns

- The four fundamental communication patterns used in gRPC-based applications are
  - unary RPC (simple RPC),
  - server-side streaming,
  - client-side streaming, and

- bidirectional streaming

Topic 11.15: Simple RPC (Unary RPC)

- In simple RPC, the client invokes a remote function of a server.
- The client sends a single request to the server and gets a single response that is sent along with status details and trailing metadata.
- Example:
- OrderManagement service for an online retail application
- rpc getOrder(google.protobuf.StringValue) returns (Order);

message Order {

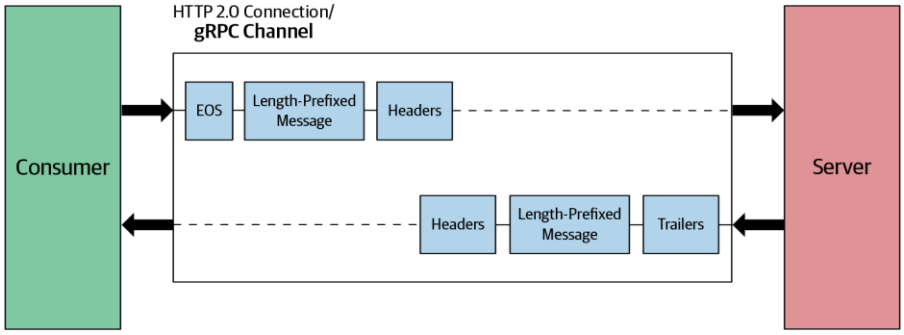
string id = 1;

repeated string items = 2;

string description = 3;

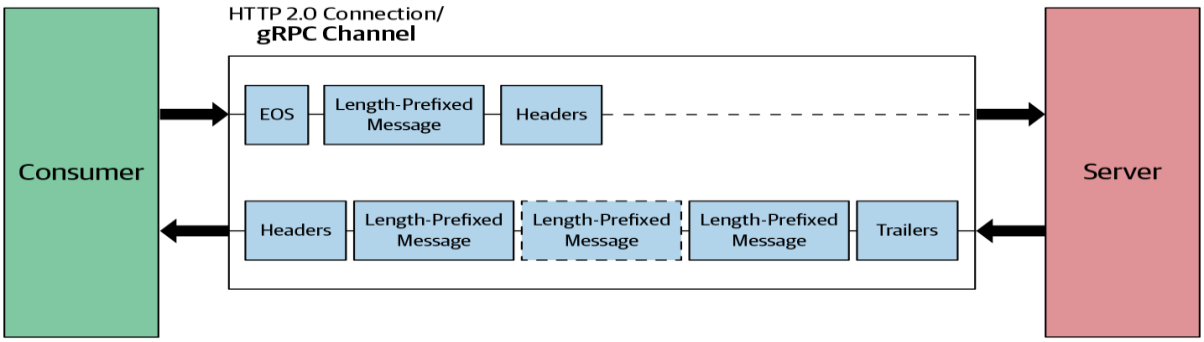
float price = 4;

}



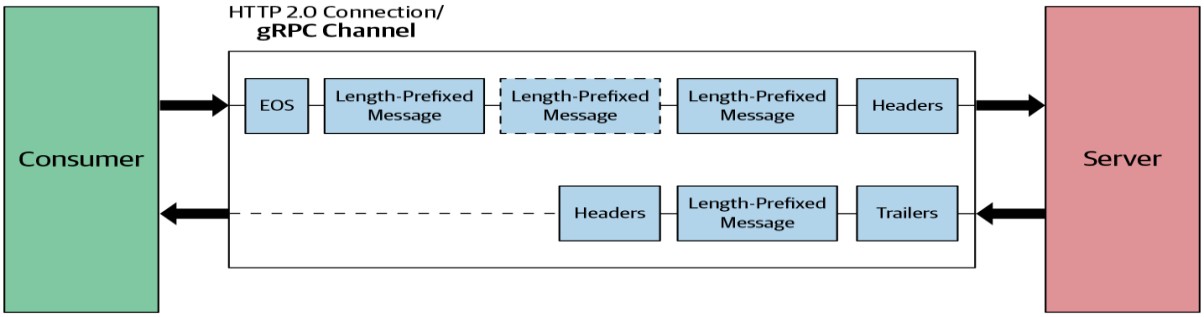
Topic 11.16: Server-Streaming RPC

- In server-side streaming RPC, the server sends back a sequence of responses after getting the client's request message.
- This sequence of multiple responses is known as a "stream."
- After sending all the server responses, the server marks the end of the stream by sending the server's status details as trailing metadata to the client.
- Example: searchOrder method of the Order Management service.
- rpc searchOrders(google.protobuf.StringValue) returns (stream Order);



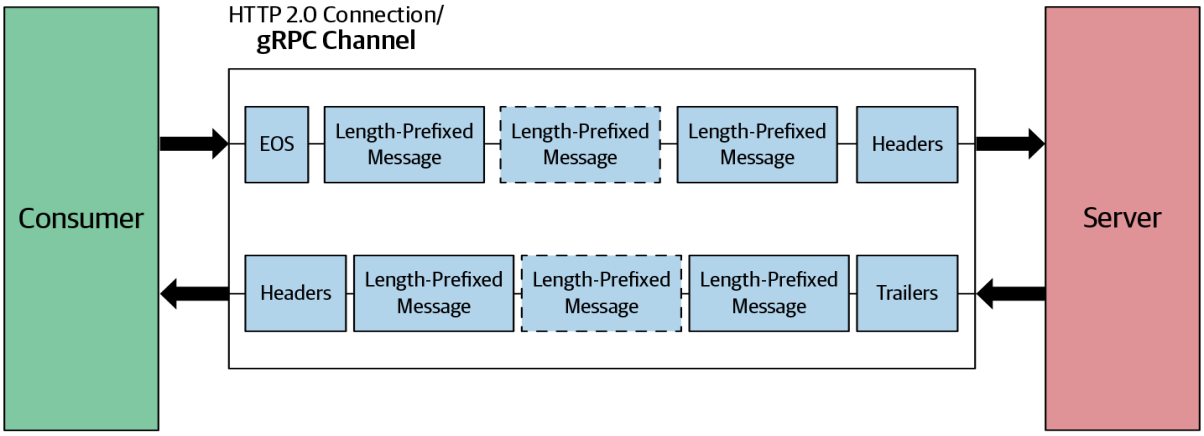
Topic 11.17: Client-Streaming RPC

- In client-streaming RPC, the client sends multiple messages to the server instead of a single request.
- The server sends back a single response to the client.
- However, the server does not necessarily have to wait until it receives all the messages from the client side to send a response.
- Example: updateOrders method of the Order Management service.
- **rpc** updateOrders(stream Order) **returns** (google.protobuf.StringValue);



Topic 11.18: Bidirectional-streaming RPC

- In bidirectional-streaming RPC, the client is sending a request to the server as a stream of messages.
- The server also responds with a stream of messages.
- The call has to be initiated from the client side,
- Example: updateOrders method of the Order Management service.
- rpc processOrders(stream google.protobuf.StringValue) returns (stream CombinedShipment);



Topic 11.19: Summary

- gRPC
- Protobuffers
- gRPC Communication Patterns

12. CS 10: GraphQL

Topic 12.1: GraphQL

- GraphQL is
  - ✓ a query language for your APIs and
  - ✓ a server-side runtime for executing queries.
- A GraphQL service is created by defining types, fields on those types and functions for each field
- A GraphQL query asks only for the data that it needs.
- GraphQL is typically served over HTTP via a single endpoint which expresses the full set of capabilities of the service.
- GraphQL services typically respond using JSON,

Topic 12.2: REST- disadvantages

- Drawbacks
  - Overfetching
  - Underfetching
- REST APIs need more flexibility.
- As the client needs change, new endpoints have to be created.
- Eg: to fetch the list of books

Topic 12.3: GraphQL

- A GraphQL operation is either
  - a query (read),
  - mutation (write), or
  - Subscription (continuous read).
- This GraphQL operation is interpreted against the GraphQL schema at the backend, and resolved with data for the frontend application.
- After a GraphQL service runs , it can receive GraphQL queries to validate and execute.
- The service first checks a query to ensure it only refers to the types and fields defined, and then runs the provided functions to produce a result.

Topic 12.4: GraphQL- Queries

GraphQL Query:

```
query {
  posts {
    title
                                author {
      name
```

```
    email
   }
  }
}
```

Topic 12.5: GraphQL Response:

```
{
 "data": {
  "posts": [
   {
    "title": "Introduction to GraphQL",
    "author": {
     "name": "John Doe",
     "email": "john.doe@example.com"
    }
   },
   {
    "title": "Getting Started with React",
                                    "author": {
     "name": "Jane Smith",
     "email": "jane.smith@example.com"
    }
   }
   // Additional posts...
  ]
 }
}
```

Topic 12.6: Queries

- GraphQL queries can traverse related objects and their fields, letting clients fetch lots of related data in one request, instead of making several roundtrips as one would need in a classic REST architecture.

Topic 12.7: Queries- Arguments and Variables

- Every field and nested object can get its own set of arguments.
- Arguments can be of many different types.
- The arguments to fields can be dynamic
- Replace the static value in the query with $variableName

- Declare $variableName as one of the variables accepted by the query
- Pass variableName: value in the separate, transport-specific (usually JSON) variables dictionary

```
{
 “personid": “1000"
}
human(id: $personid)
```

Topic 12.8: Queries- Arguments and Variables

Query:

```
{
 human(id: "1000") {
   name
   height(unit: FOOT)
  }
}
```

Response:

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

Queries

- Default values
- Directives
- Reusable units called *fragments*

Mutations

- Mutations are used to modify data on the server.
- To write new data, we use *mutations*.
- Mutations are defined like queries.
- The Mutation is a root object type.

```
mutation {
  addUser(name: "Alice", age: 30) {
    id
```

```
  name
  age
 }
}
```

Topic 12.9: Schemas and Types

- Every GraphQL service defines a set of types describing the possible data you can query for that service.
- Then, when queries come in, they are validated and executed against that schema.

Schema

- Schema define the data types that your API will expose.
- GraphQL APIs are defined by a schema, which serves as a contract between the client and the server.
- The schema specifies the types of data that can be queried and the relationships between them.
- GraphQL schema documents are text documents that define the types available in an application

Types

- The core unit of any GraphQL Schema is the type.
- These types represent the structure of the data available in the API.
- A type has *fields* that represent the data associated with each object.
- Each field returns a specific type of data.
- A schema is a collection of type definitions

```
type User {
  id: ID!
  name: String!
  age: Int!
 }
```

Topic 12.10: Schema and Types

- Two types are special within a schema
- Query and
- Mutation
- Every GraphQL service has a query type and may or may not have a mutation type.
- They are special because they define the *entry point* of every GraphQL query

```
const schema = buildSchema(`
 type User {
```

```
  id: ID!
  name: String!
  age: Int!
 }

 type Query {
  getUser(id: ID!): User
  getUsers: [User]
 }

 type Mutation {
  addUser(name: String!, age: Int!): User
 }
`);
```

Topic 12.11: Resolvers

- Resolvers are functions that determine how data is retrieved or modified.
- They map to the types and fields defined in the schema.
- Resolver functions return data in the type and shape specified by the schema.
- Resolvers can fetch or update data from a REST API, database, or anyother service.

Topic 12.12: GraphQL Service

- A GraphQL service can be written in any programming language,
- It is conceptually split into two major parts,
    - structure and
    - Behavior
- The structure is defined with a strongly typed schema.
- The behavior is naturally implemented with functions and are called resolver functions.

Topic 12.13: Graphs

- In GraphQL, the foundation is a graph-based approach to modeling business domain.
- By employing GraphQL, the business domain is constructed as a graph,
- Graphs are used formally to represent a collection of interconnected objects.
- Schema outlines different types of nodes and their connections or relationships.

Examples

- Facebook is an undirected graph.
- Think of your GraphQL schema as an expressive shared language for your development team and end-users.

- Creating a robust schema involves examining the everyday language used to describe your business.

Topic 12.14: GraphQL language

- At the core of a GraphQL communication is a request object.
- The source text of a GraphQL request is often referred to as a document.
- A document contains text that represents a request through operations like queries, mutations, and subscriptions.
- In addition to the main operations, a GraphQL document text can contain fragments that can be used to compose other operations.

Topic 12.15: Summary

- GraphQL
- Schemas and Types
- Queries and Mutations
- Resolvers

13. CS11 Security

Topic 13.1: Agenda

- Basic Authentication
- API Keys
- JWT
- OAuth

Topic 13.2: Security

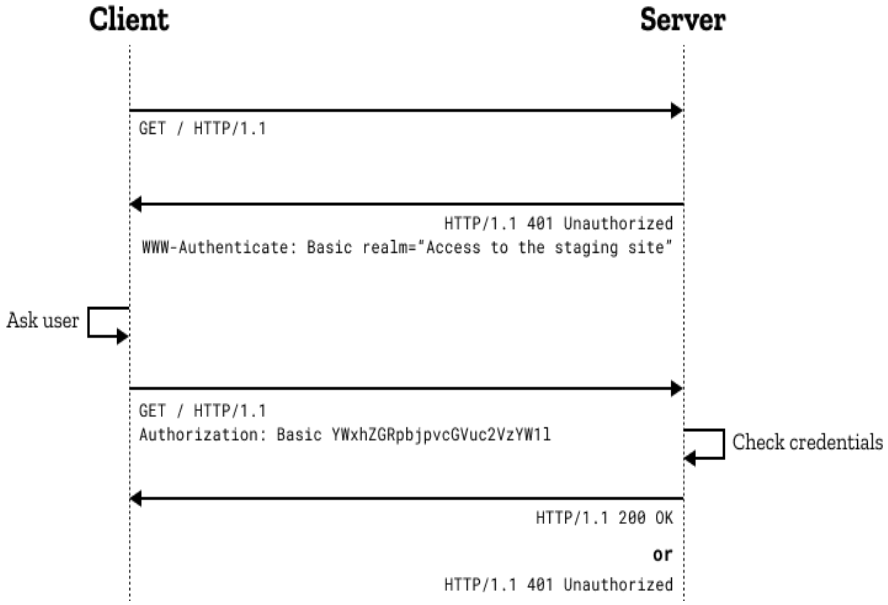
- Authentication and authorization are two foundation elements of security:
- Authentication is the process of verifying who a user is.
- Authorization is the process of verifying what they have access to.

Topic 13.3: Basic Authentication

- HTTP provides a general framework for access control and authentication.
- In basic HTTP authentication, a request contains a header field in the form of

Authorization: Basic <credentials>

- The credentials is the Base64 encoding of username and password joined by a single colon
- Basic authentication is typically used in conjunction with HTTPS to provide confidentiality.



Topic 13.4: Basic Authentication

- Here are some reasons why it is still used:
  - Simplicity
  - Compatibility
  - Statelessness
- Limitations

- Security
- Single Factor
- Risk of Credential Exposure

Topic 13.5: API Keys

- An **API key** is a unique identifier used to authenticate and authorize the access to an **API**
- Instead of a username and password, the client application is issued a unique API key, typically a long alphanumeric string.
- The API key is sent in the HTTP request as a parameter in the query string or the request headers
- (e.g., api_key=your_api_key or Authorization: API-Key your_api_key).
- API keys are commonly used for machine-to-machine communication or applications interacting with the API.
- Some APIs use API keys to enforce rate limits.
- **Security**: Treat API keys as sensitive information. Avoid hardcoding them in client-side code or exposing them publicly.
- **Rotation**: Regularly rotate API keys to enhance security. You can invalidate a key and issue a new one if a key is compromised.
- **Scopes**: Consider using different keys for different purposes (e.g., read-only vs. administrative access).
- **HTTPS**: Always use HTTPS to transmit API keys securely.
- Examples: **Google Maps API, Cloud Services**:

Topic 13.6: Token-based authentication system

- Token-based authentication allows users to verify their identity, and in return receive a unique access token.
- Token-based authentication is different from traditional password-based technique.
- In stateless communication, each request that the user makes to the server contains all the necessary information for authentication, typically in the form of token.
- The server validates the token and responds accordingly for each request.

Topic 13.7: Types Of Tokens

- **Opaque tokens**
- The opaque token is a random, unique string of characters the authorization server issues.
- The opaque token does not pass any identifiable information
- To validate the token and retrieve the information on the token and the user, the resource server calls the authorization server and requests the token introspection.
- **Structured token:**
- Its format is well-defined so the resource server can decode and verify the token without calling the authorization server.
- JWT is a structured token

Topic 13.8: JSON Web Tokens (JWT)

- JSON Web Tokens (JWTs) are a format of tokens used in web development and security.
- JWT is a standard way to securely represent claims, such as user identity and roles, between two parties.
- A JWT has a payload, which is a JSON object that contains information about the user, such as their identity and roles, and other metadata, such as an expiration date.
- It's signed with a secret that's only known to the creator of the JWT.
- The secret ensures a malicious third party can't forge or tamper with a JWT.
- JSON Web Tokens consist of three parts separated by dots (.), which are:
    - Header
    - Payload
    - Signature
- Therefore, a JWT typically looks like the following: xxxxx.yyyyy.zzzzz
- **Header**
- The header typically consists of two parts: the type of the token, which is JWT, and the hashing algorithm such as HMAC SHA256 or RSA.
- Then, this JSON is Base64Url encoded to form the first part of the JWT.
- **Payload**
- The second part of the token is the payload, which contains the claims.
- Claims are statements about an entity (typically, the user) and additional metadata.
- The payload is then **Base64Url** encoded to form the second part of the JWT.

```
Header:
{
"alg": "HS256",
  "typ": "JWT"
}
```

```
Payload:
{
    "sub": "1234567890",
    "name": "John Doe",
     "admin': true
}
```

Topic 13.9: JSON Web Tokens (JWT)

- **Signature**

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

HMACSHA256(

base64UrlEncode(header) + '.' +

base64UrlEncode(payload),

secret)

- The signature is used to verify that the sender of the JWT
- The output is three Base64 strings separated by dots

Topic 13.10: Uses of JWT

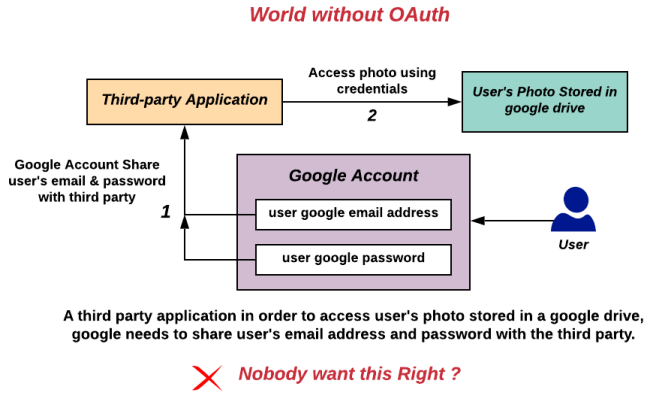
- Information Exchange
- Single Sign on
- Authentication and Authorization

14. CS12 OAuth

Topic 14.1: OAuth

- OAuth (Open Authorization) is an open standard for access delegation.
- It is commonly used as a way for internet users to grant websites or applications access to their information on other websites.
- It specifies a process for resource owners to authorize third-party access to their server resources without providing credentials.
- OAuth works over HTTPS

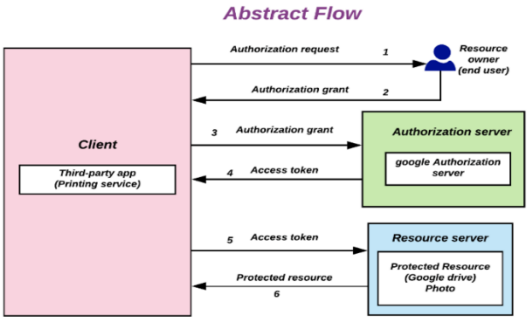
Topic 14.2: Without OAuth



Topic 14.3: OAuth

- OAuth is a delegated authorization framework for REST/APIs.
- It enables apps to obtain limited access (scopes) to a user's data without giving away a user's password.
- It decouples authentication from authorization and supports multiple use cases addressing different device capabilities.
- It supports server-to-server apps, browser-based apps, mobile/native apps, and consoles/TVs.
- Analogy: If you have a hotel key card, you can access your room.
- How do you get a hotel key card?
- You have to do an authentication process at the front desk to get it.
- After authenticating and obtaining the key card, you can access the room and resources permitted across the hotel.
- Similarly ,App requests authorization from User

Topic 14.4: OAuth Flow



Topic 14.5: OAuth Central Components

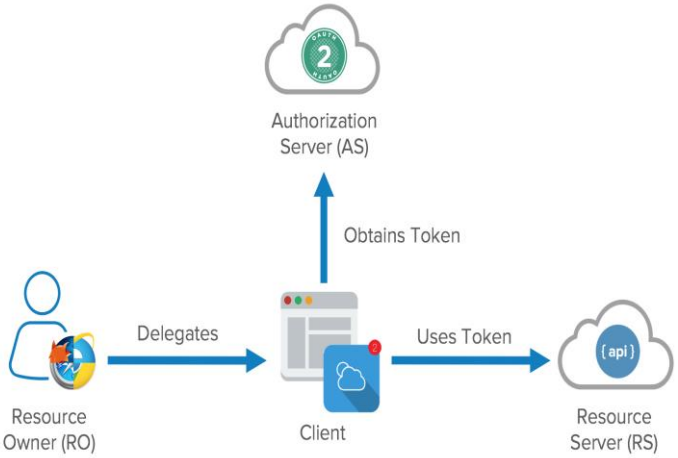
- OAuth is built on the following central components:
    - Scopes and Consent
    - Actors
    - Tokens
    - Flows

Topic 14.6: Scopes

- Scopes are what you see on the authorization screens when an app requests permissions.
- They're bundles of permissions asked for by the client when requesting a token.
- These are coded by the application developer when writing the application.

Topic 14.7: Actors

- The actors in OAuth flows are as follows:
- **Resource Owner**: owns the data in the resource server.
- **Resource Server**: The API which stores data the application wants to access
- **Client**: the application that wants to access your data
- **Authorization Server**: The main engine of OAuth



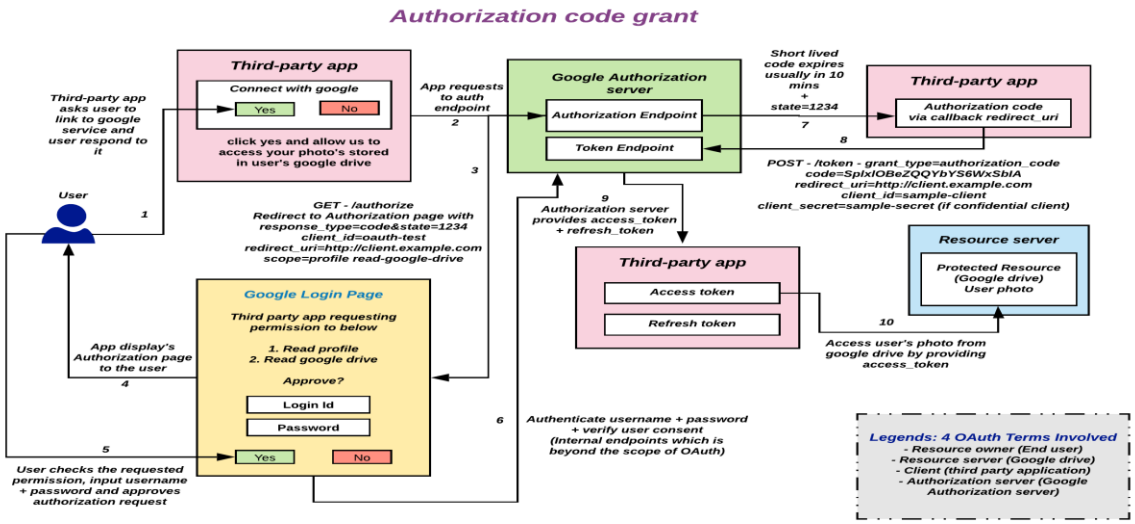
Topic 14.8: Tokens

- Access tokens are the token the client uses to access the Resource Server (API).
- They're meant to be short-lived.
- Refresh Tokens can be used to get new tokens
- The OAuth spec doesn't define what a token is
- Usually JWT is used
- Tokens are retrieved from endpoints on the authorization server.
- The two main endpoints are the authorize endpoint and the token endpoint.

Topic 14.9: Flows

- OAuth framework specifies several grant types for different use cases.
- OAuth grant types
    - Authorization Code
    - Client Credentials
    - Implicit Flow
    - Resource Owner Password Flow

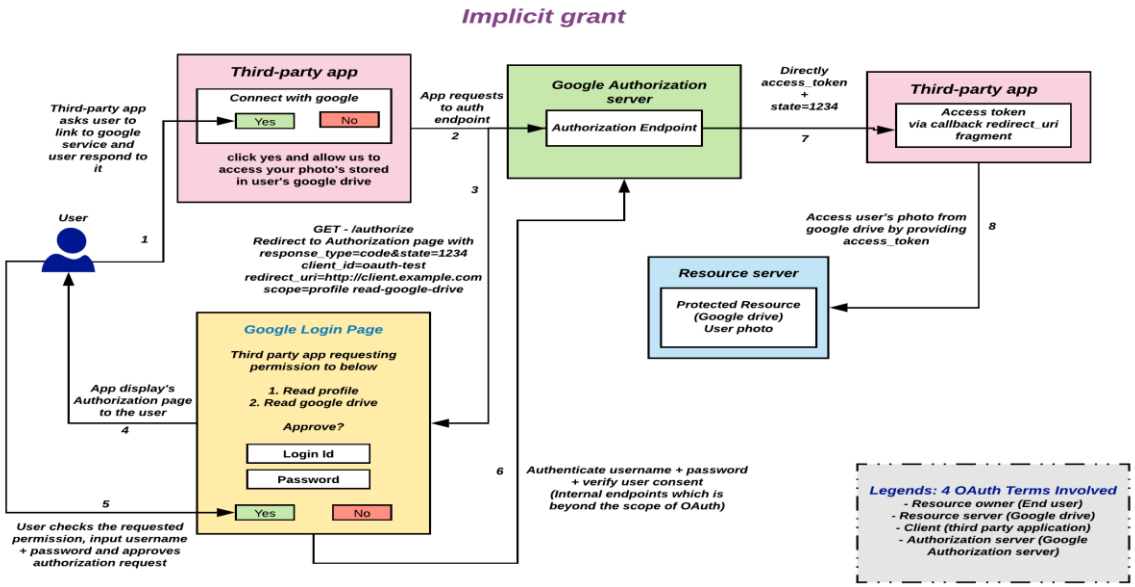
Topic 14.10: Authorization code



Topic 14.11: Authorization code flow

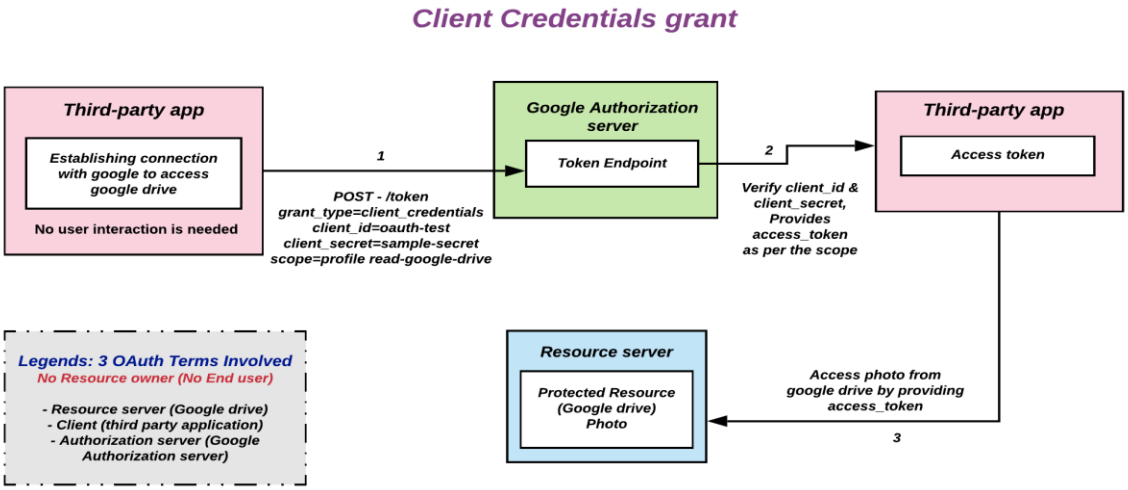
- Use Case: Regular web apps executing on a server.
- Example: A web application that needs to securely retrieve an access token.
- The client (web app) exchanges an authorization code for an access token. It's considered safe because the token is passed directly to the server without going through the user's browser.

Topic 14.12: Implicit Flow



- An access token is returned directly from the authorization request.
- It typically does not support refresh tokens.
- Since everything happens on the browser, it's the most vulnerable to security threats.
- An SPA is a good example of this flow's use case.

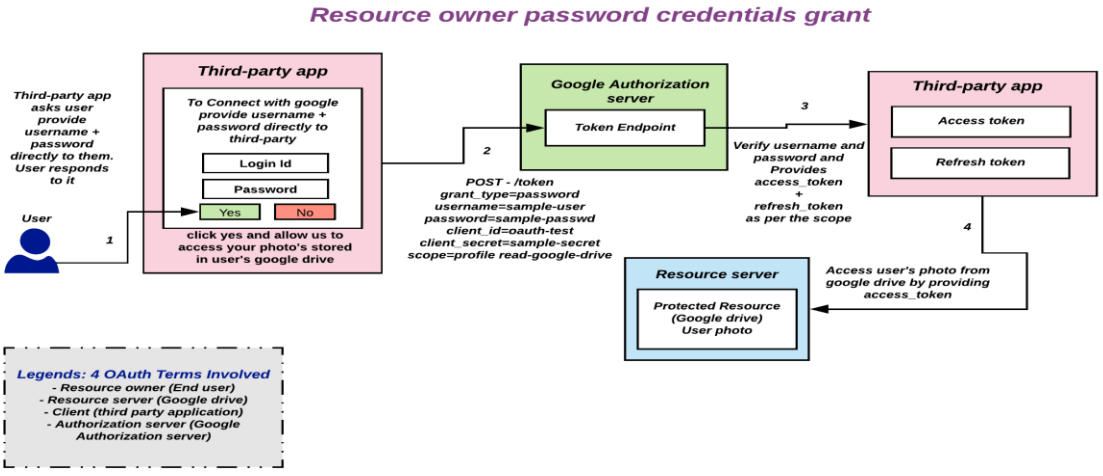
Topic 14.13: Client Credentials flow



- For server-to-server scenarios, a Client Credential Flow is used
- In this scenario, the client application is a confidential client that's acting on its own.
- It's a back channel only flow to obtain an access token using the client's credentials.
- It supports shared secrets or assertions as client credentials
- Use Case: Machine-to-machine authorization where no end-user interaction is needed.
- Example: A cron job that imports data to a database using an API.
- How It Works: The client (e.g., the cron job) directly obtains an access token from the authorization server using its client ID and client secret.

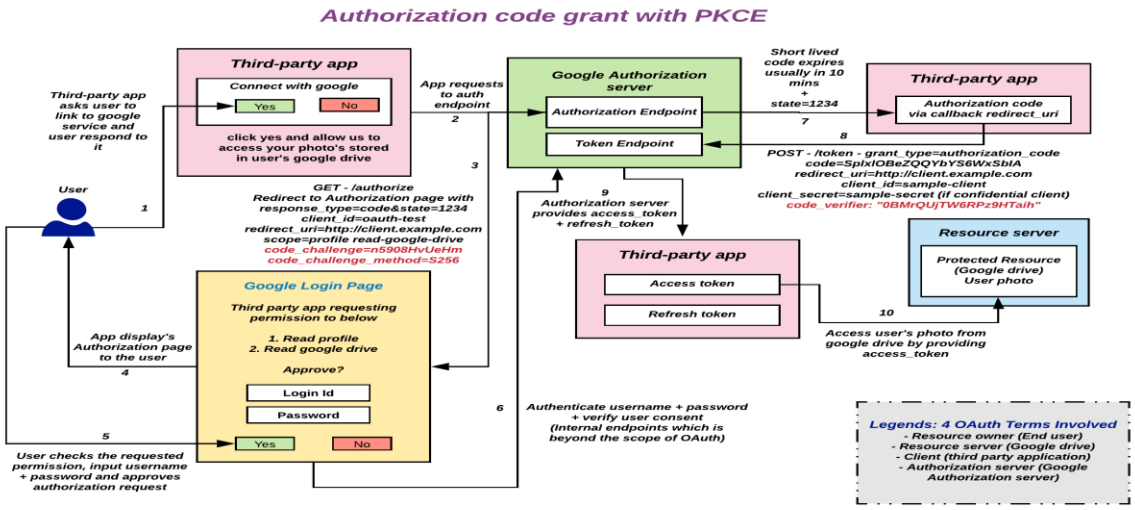
Topic 14.14: Resource Owner Password Flow

- It's a legacy grant type for native username/password apps like desktop applications.
- In this flow, you send the client application a username and password and it returns an access token from the Authorization Server.

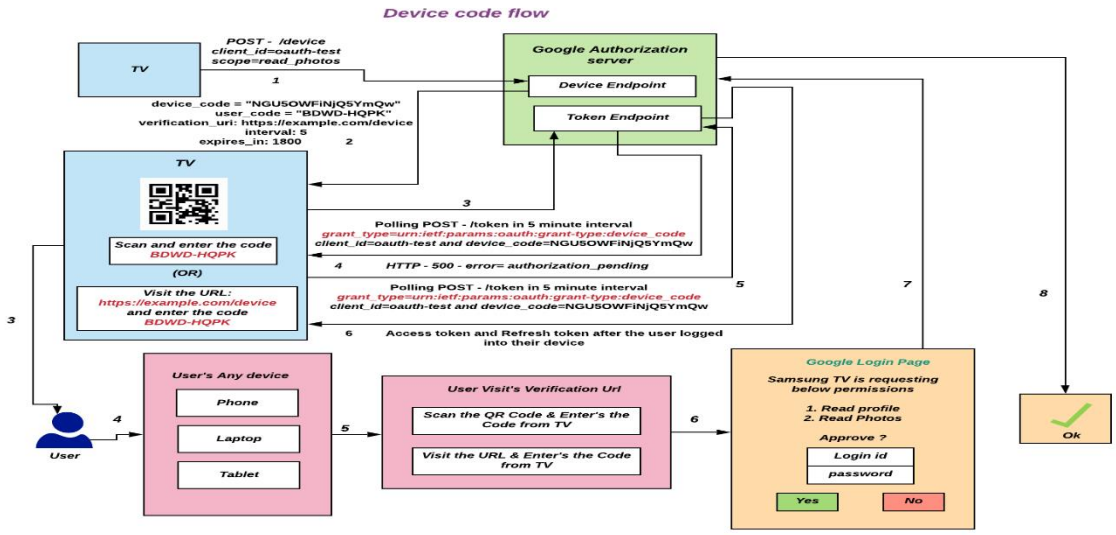


Topic 14.15: Authorization code with PKCE

- This flow is an extension to Authorization grant flow.
- Authorization code grant is vulnerable to authorization code interception attacks when used with public clients
- Proof Key for Code Exchange(PKCE)



Topic 14.16: Device Code Flow

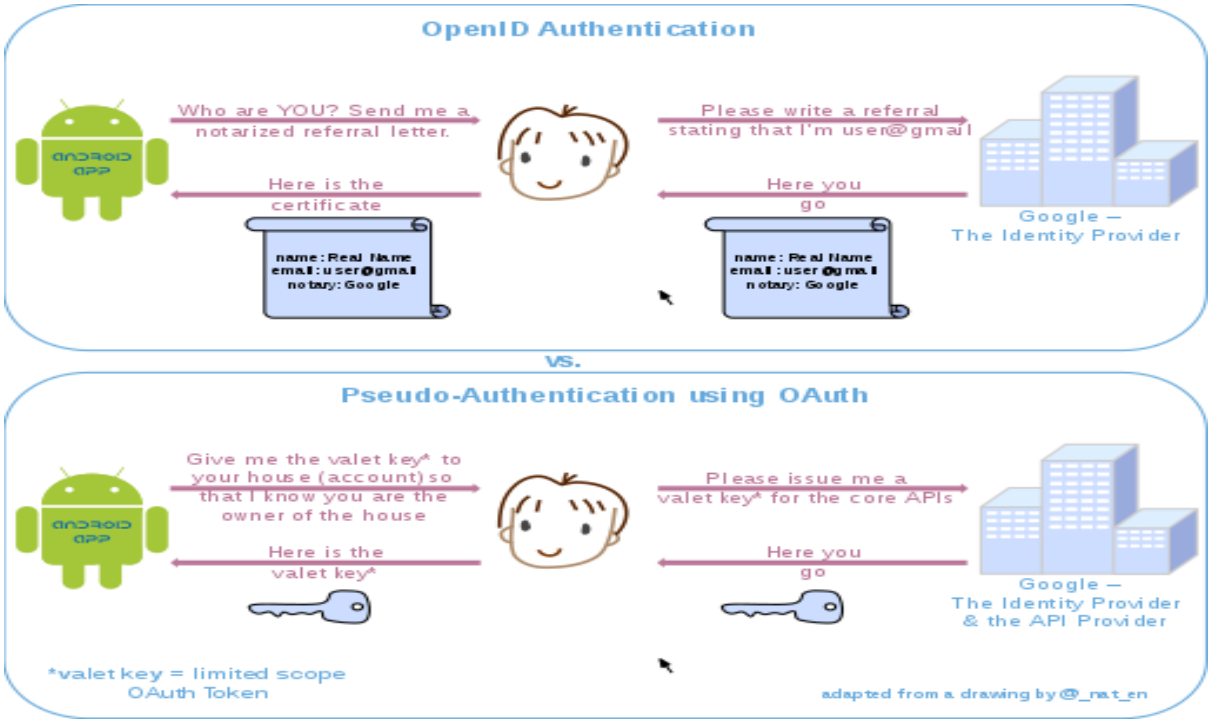


Topic 14.17: pseudo-authentication using OAuth

- OAuth is an authorization protocol, rather than an authentication protocol.
- OAuth does not provide user's information via an access token
- Access tokens are meant to be opaque.
- They're meant for the API, they're not designed to contain user information.
- Custom Hacks were used to fill this gap

- Using OAuth on its own as an authentication method may be referred to as pseudo-authentication

Topic 14.18: OpenID vs OAuth



Topic 14.19: OpenID Connect

- OAuth is directly related to OpenID Connect (OIDC).
- OIDC is an authentication layer built on top of OAuth 2.0.
- OpenID Connect (OIDC) extends OAuth 2.0 with a new signed id_token for the client and a UserInfo endpoint to fetch user attributes
- OpenID Connect is the standard for identity provision on the Internet.
- What it adds:
    - ID token
    - User endpoint to get more userinfo
    - Standardized
- Its formula for success: simple JSON-based identity tokens (JWT), delivered via OAuth 2.0 flows that fit web, browser-based and native / mobile applications.

Topic 14.20: References

- Demystifying OAuth 2.0 - A Tutorial & Primer :: Devansvd — Personal website
- https://blog.postman.com/pkce-oauth-how-to/
- https://auth0.com/docs/get-started/authentication-and-authorization-flow/which-oauth-2-0-flow-should-i-use

15. CS 13 Front End

Topic 15.1 Frontend technologies

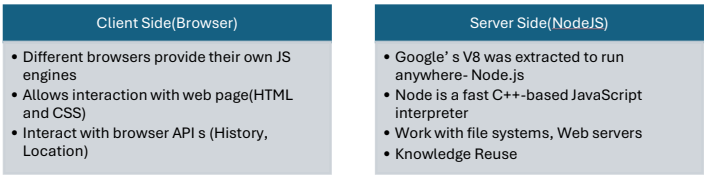
- What happens when you load your webpage in the browser?
- The code (the HTML, CSS, and JavaScript) are running inside an execution environment (the browser).
- HTML is the markup language that we use to structure and give meaning to our web content.
- CSS is a language of style rules that we use to apply styling to our HTML content, for example, setting background colors and fonts and laying out our content in multiple columns.
- JavaScript is a scripting language that enables you to create dynamically updating content and adds interactivity to your webpage.

Topic 15.2 Javascript

- JavaScript is the programming language of the web.
- JavaScript is a single-threaded interpreted language.
- Every browser has its own JavaScript engine. Google Chrome has the V8 engine, Mozilla Firefox has SpiderMonkey.
- The core client-side JavaScript language consists of some common programming features like variables, objects, functions etc.,

Topic 15.3 Host Environment

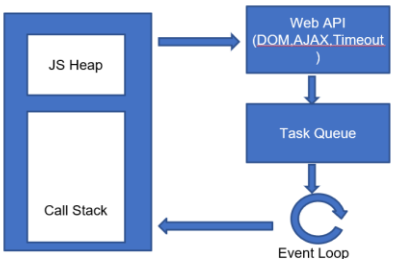
- Browsers- Initially only implemented in web browsers
- Now it can be used on Server Side



Topic 15.4 The Critical Rendering Path(Recap)

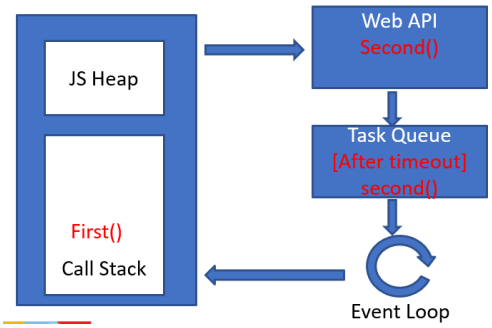
Topic 15.5 Execution in Javascript

- JavaScript is a single-threaded programming language.
- An Execution Context is an abstract concept of an environment where the JavaScript code is evaluated and executed.
- **The JavaScript Engine** consists of two main components:
- Memory Heap: this is where the memory allocation happens
- Call Stack: this is where your stack frames are as your code executes. As its name implies, the call stack has a LIFO (Last in, First out) structure, which stores all the execution context created during the code execution.
- Synchronous Execution




```
<script>
    const second = () => {
        console.log('Hello there!');
    }
    const first = () => {
        console.log('Hi there!');
        second();
        console.log('The End');
    }
    first();
</script>
```

- Asynchronous Execution




```
<script>
    const second = () => {
        console.log('Hello there!');
    }
    const first = () => {
        console.log('Hi there!');
        setTimeout(second, 1000);
        console.log('The End');
    }
    first();
</script>
```

Topic 15.6 WebAPIs

Topic 15.7 Client-side JavaScript API

- Client-side JavaScript has many APIs available.
- They are not part of the JavaScript language itself.
- But they are built on top of the core JavaScript language.
- Client-side JavaScript API s fall into two categories
    - Browser APIs
    - Third Party APIs

Topic 15.8 Browser API

- A Browser API can extend the functionality of a web browser.
- All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.
- For example, the Web Audio API, Geolocation API

Topic 15.9 Third-party APIs

- **Third-party APIs** are not built into the browser by default.
- Third-party APIs are constructs built into third-party platforms (e.g. Google Maps API, Facebook APIs)
- They allow you to use some of those platform's functionality in your own web pages

Topic 15.10 Common APIs

- **APIs for manipulating documents**
    - Example: DOM API
- **APIs that fetch data from the server**
    - Example: Fetch API
- **APIs for drawing and manipulating graphics**
    - Example Canvas API
- **Audio and Video APIs**
    - Example: Web Audio API
- **Device APIs**
    - Example: Geolocation API
- **Client-side storage APIs**
    - Example: Web Storage API

Topic 15.11 Client-side storage

- There are numerous methods for storing data locally in the users' browser
    - Cookies

- Web Storage (Local and Session Storage)
- IndexedDB
- Centralized data store ; State management libraries can be used.

Topic 15.12 Cookies

- A cookie is a small piece of data that a server sends to the user's web browser.
- The browser may store it and send it back with the next request to the same server.
- It remembers stateful information for the stateless HTTP protocol.
- They're the earliest form of client-side storage commonly used on the web.

Topic 15.13 Web Storage API

- The Web Storage API provides mechanisms by which browsers can store key/value pairs
- The two mechanisms within Web Storage are as follows:
    - sessionStorage maintains a separate storage area for each given origin that's available for the duration of the page session
    - localStorage does the same thing, but persists even when the browser is closed and reopened.
    - The two types of storage areas are accessed through global objects named “window.localStorage” and “window.sessionStorage”.
- Data is stored as key/value pairs, and all data is stored in string form.
- Data is added to storage using the setItem() method.
- setItem() takes a key and value as arguments.
- If the key does not already exist in storage, then the key/value pair is added.
- If the key is already present, then the value is updated.
- sessionStorage.setItem("foo", 3.14);
- localStorage.setItem("bar", true);

Topic 15.14 Reading Stored Data

- To read data from storage, the getItem() method is used.
- getItem() takes a lookup key as its sole argument. If the key exists in storage, then the corresponding value is returned.
- If the key does not exist, then null is returned.
- var number = sessionStorage.getItem("foo");
- var boolean = localStorage.getItem("bar");

Topic 15.15 Removing Stored Data

- To delete individual key/value pairs from storage, the removeItem() method is used.
- The removeItem() method takes the key to be deleted as its only parameter.
- If the key is not present then nothing will happen.
- sessionStorage.removeItem("foo");

- localStorage.removeItem("bar");

Topic 15.16 The storage Event

- A user can potentially have several instances of the same site open at any given time.
- Changes made to a storage area in one instance need to be reflected in the other instances for the same domain.
- The Web Storage API accomplishes this synchronization using the “storage” event.
- When a storage area is changed, a “storage” event is fired for any other tabs/windows that are sharing the storage area.
- Note that a “storage” event is *not* fired for the tab/window that changes the storage area.

Topic 15.17 Indexed DB

- IndexedDB is a transactional database embedded in the browser.
- The database is organized around the concept of collections of JSON objects similar to NoSQL databases
- IndexedDB is useful for applications that store a large amount of data (for example, a catalog of DVDs in a lending library) and applications that don't need persistent internet connectivity to work (for example, mail clients, to-do lists, and notepads).
- Each IndexedDB database is unique to an origin
- IndexedDB is built on a transactional database model.
- Database - This is the highest level of IndexedDB. It contains the object stores, which in turn contain the data you would like to persist.
- Object store - An object store is an individual bucket to store data. Similar to tables in traditional relational databases.
- Operation - An interaction with the database.
- Transaction - A transaction is wrapper around an operation, or group of operations, that ensures database integrity.

Topic 15.18 Node Ecosystem

Topic 15.19 Node JS Ecosystem

- **Node.js Core**
- Node.js was built using the V8 JavaScript Engine
- Compiles JavaScript to machine code
- **Node.js Runtime**
- event-driven, non-blocking I/O model
- **NPM (Node Package Manager)**
- Command-line tool
- Manages Dependencies
- **Express.js**
- Express.js is a popular, minimalistic web application framework for Node.js.

- **Webpack** is a powerful **module bundler**
- **Webpack** is a tool that takes your **source code** (JavaScript, CSS, images, etc.) and **bundles** it into a single file (or multiple files) that can be easily deployed to a web server.
- Its main purpose is to **optimize** and **organize** your code, making it

Topic 15.20 Transpiling

- Transpiling refers to the process of converting ECMAScript 6 (ES6) code or the latest code into an older version of JavaScript that is more widely supported by browsers and environments.
- **Babel** is the most popular transpiler for ES6.

Topic 15.21 Resources

- https://www.jsv9000.app/
- https://eloquentjavascript.net/11_async.html
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop

16. CS 14 ReactJS

Topic 16.1: React - Introduction

- React is a JavaScript library for rendering user interfaces (UI).
- Open-source
- It is maintained by Meta (formerly Facebook) and a community of individual developers and companies.

Topic 16.2: Features

- What made React stand out and popular?
    - React adheres to the declarative programming paradigm
    - React lets you break down the UI into reusable components
    - Streamlines the process of building and composing components
    - React uses a virtual DOM
    - It is easy to learn and use

>> Why react is declarative programming?

Aspect	How React Makes It Declarative
UI Definition	JSX lets you define what the UI should look like.
State Management	The UI automatically updates when state or props change.
DOM Updates	React's Virtual DOM abstracts away manual DOM handling.
Event Handling	Events are defined declaratively via props like `onClick`.
Component Hierarchy	Components describe the UI at a high level of abstraction.

>> What is Virtual DOM?

- In the traditional method, the entire DOM is recreated whenever an event occurs, causing the whole page to re-render.

- However, in React, a virtual copy of the DOM is created with the changes caused by the event.

- React then compares this virtual DOM with the original DOM and updates only the differences between them.

- As a result, only the specific component gets updated and re-rendered instead of the entire page.

Topic 16.3: Component based Approach

- React is based on a component-based architecture that allows developers to break down their user interface into small, reusable components.
- This makes it easier to manage and maintain complex UI
- Developers can focus on developing and testing individual components separately.
- Each component consists of well-defined functionality that can be inserted into an application without requiring modification of other components

Topic 16.4: Declarative- What React Simplifies

- Consider the task of adding a element

```
const target = document.getElementById("target");
const wrapper = document.createElement("div");
const headline = document.createElement("h1");

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);


const { render } = ReactDOM;
const Welcome = () => (
  <div id="welcome">
    <h1>Hello World</h1>
  </div>
);
render(<Welcome />,
document.getElementById("target"));
```

Topic 16.5: Virtual DOM

- React uses a virtual DOM, which is a **lightweight representation** of the actual DOM.
- React updates the **virtual DOM** instead of the actual DOM
- React then **compares** the virtual DOM with the actual DOM and only updates the necessary parts of the UI
- Every time the DOM changes, the browser has to do two intensive operations: repaint and reflow
- Whenever a change is required, React marks that Component as **dirty.**
- React updates the Virtual DOM relative to the components marked as dirty
- React batches much of the changes and performs a **unique update** to the real DOM.
- Repaint and reflow the browser must perform to render the changes are executed just once.

Topic 16.6: Important Javascript features

- Data types
- Using var, let and const
- Conditionals and Loops
- Using objects, arrays and functions
- ES6 Arrow functions
- In-built functions such as map(), forEach() and promises.
- Destructuring Arrays and Objects
- Modules

>> var, let and const:

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted with `undefined`	Hoisted but in the TDZ	Hoisted but in the TDZ
Re-declaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed
Immutability	No	No	Only the binding is immutable

>> ES6 Arrow function:

Feature	Details
Syntax	`const add = (a, b) => a + b;`
`this` **Binding**	Lexically inherits `this` from the enclosing scope.
Implicit Return	Single-expression functions return the result without `return` keyword.
No `arguments`	Does not have an `arguments` object; use rest parameters (`...args`).
Use Case	Ideal for short functions, callbacks, and functional programming tasks.

>> map, forEach and Promises in JS:

Feature	`map`	`forEach`	Promises
Purpose	Creates a new array by transforming each element	Executes a function on each array element	Manages asynchronous operations
Return Value	Returns a new array	Returns `undefined`	Returns a `Promise` object
Callback	Mandatory, processes each element	Mandatory, processes each element	Handles success (`then`) and failure (`catch`)
Chaining	Can chain methods like `.filter()` or `.reduce()`	Cannot be chained	Supports chaining (`.then().catch()`)
Use Case	Transforming array data	Side effects, like logging or mutations	Handling async tasks like API calls or file I/O
Example	`[1, 2, 3].map(x => x * 2); // [2, 4, 6]`	`[1, 2, 3].forEach(x => console.log(x)); // 1, 2, 3`	`fetch('url').then(res => res.json()).catch(err => console.error(err));`

>> Destructuring Arrays and Objects

Feature	Destructuring Arrays	Destructuring Objects
Purpose	Extract values from arrays into variables	Extract values from object properties into variables
Syntax	`const [a, b] = [1, 2];`	`const { name, age } = { name: 'Alice', age: 25 };`
Default Values	`const [a = 5] = [];`	`const { name = 'Default' } = {};`
Nested Destructuring	`const [[a], b] = [[1], 2];`	`const { address: { city } } = { address: { city: 'NY' } };`
Rest Syntax	`const [a, ...rest] = [1, 2, 3];`	`const { name, ...rest } = { name: 'Alice', age: 25 };`
Use Case	Simplifies array value assignment	Simplifies object property extraction
Example	`const [a, b] = [1, 2]; // a = 1, b = 2`	`const { name, age } = { name: 'Alice', age: 25 }; // name = 'Alice', age = 25`

Topic 16.7: Create react app

- Create React App (CRA) is officially deprecated by the React team, primarily because it has limitations compared to newer, more flexible tools
- Lack of Configurability
- Outdated Technology
- Maintenance Overhead

Topic 16.8: Alternatives to Create React App

- **Vite**: Known for its speed, Vite is optimized for modern JavaScript and React projects. It provides near-instant startup, fast hot-module replacement (HMR), and excellent configurability. Vite has become very popular for both small and large-scale React applications.
- **Next.js**: Next.js is versatile and can handle single-page applications. It offers built-in routing, API routes, and server-side rendering (SSR) options, making it ideal for production-level React apps.
- Remix is a framework that emphasizes full-stack React applications with a focus on web standards and optimization for faster performance. It’s a good choice for projects where routing and server-side data fetching are important.
- Parcel is a zero-config bundler that’s easy to use and has great performance. It's a viable option for those who prefer minimal setup while still needing good development speed.

Topic 16.9: Folder Structure

```
my-app/
  README.md
  node_modules/
  package.json
  public/
    index.html
    favicon.ico
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```

Topic 16.10: React Elements

- The elements that make up an HTML document become DOM elements when the browser loads HTML and renders the user interface.
- The browser DOM is made up of DOM elements.
- Similarly, the React DOM is made up of React elements.
- A React element is a description of what the actual DOM element should look like.
- Create a React element to represent an h1 using React.createElement:
    - React.createElement("h1", { id: “listitem-0” }, “Web Technologies");
    - During rendering, React will convert this element to an actual DOM element:
    - <h1 id=“listitem-0">Web Technologies</h1>

Topic 16.11: ReactDOM

- ReactDOM contains the tools necessary to render React elements in the browser.
- ReactDOM contains the render method.
- const listem1 = React.createElement("h1", { id: "listitem-0" }, "Web Technologies");
- ReactDOM.render(listem1, document.getElementById("root"));

Topic 16.12: JSX

- const element = <h1 id=“item1”>Web Technologies</h1>;
- const element=React.createElement(‘h1’, {id:’item1’}, ‘Web Technologies’);
- It creates Javascript object.
    - const name = 'John Doe';
    - const element = <h1>Hello, {name}</h1>;
    - ReactDOM.render(
    - element,
    - document.getElementById('root')
    -);

Topic 16.13: Changes to be noted

- **class becomes className**
    - Due to the fact that JSX is JavaScript, and class is a reserved word
    -
    - for which is translated to htmlFor
- **Inline Style : CSS in React**
- **Instead of accepting a string containing CSS properties, the JSX style attribute only accepts an object**
    - var divStyle = {
    - color: 'white'
    - }
    - ReactDOM.render(Hello World! , mountNode)

Topic 16.14: Mapping Arrays with JSX

- To render multiple JSX elements in React, you can loop through an array with the .map() method and return a single element.
- function courseListItems() {
- const courses = [“Web Technologies", “Java", “C++"];
- return courses.map((course) => <li key={course}>{course});
- }
- add a unique key to identify each list item uniquely

Topic 16.15: React Fragments

- We render Adjacent elements (two siblings) using a React fragment.
- function listitem({ name }) {
- return (
- <h1> {name}</h1>
- This is the first list item.
-);
- }

Topic 16.16: Components

- Components let you split the UI into independent, reusable pieces.
- Components allow us to reuse the same structure with different pieces of data

Topic 16.17: Types

- Functional Components
  - function Welcome(props) {
  - return <h1>Hello, {props.name}</h1>;
  - }
- Class Components
  - class Welcome extends React.Component {
  - render() {
  - return <h1>Hello, {this.props.name}</h1>;
  - }
  - }

Topic 16.18: Rendering a Component

- function Course(props) {
- return <h1>Course: {props.name} , {props.credits} </h1>;
- }
- const element = <Course name="Java" credits="4" />;
- ReactDOM.render(
- element,
- document.getElementById('root')
-);

Topic 16.19: Props

- Props is how Components get their properties.
- Starting from the top component, every child component gets its props from the parent.

- In a function component, props are available by **adding props as the function argument**
- In a class component, props are passed by default.
- They are accessible as **this.props** in a Component instance.
- When initializing a component, pass the **props** in a way similar to **HTML attributes**:
- Props are **Read-Only**
- Whether you declare a component as a function or a class**, it must never modify its own props.**

Topic 16.20: Presentational vs container components

- In React components are often divided into 2 big buckets:
    - presentational components and
    - container components.
- **Presentational components are mostly concerned with generating some markup to be outputted.**
- They don't manage any kind of state, except for state related the presentation.
- **Container** components are mostly concerned with the **"backend" operations**.
- They might **handle the state of various sub-components**.
- They might **wrap several presentational components**.
- They might **interface with Redux**.
- Presentational components are concerned with the look,
- container components are concerned with making things work.

Topic 16.21: State

- React Class components has a built-in state object.
- The state object is where you store property values that belongs to the component.
- When the state object changes, the component re-renders.

Topic 16.22: State in Class Component

- class JobList extends Component {
- constructor(props) {
- super(props);
- this.state = {cart: [{name:'carservice'}]
- };
- }

Topic 16.23: Points to be Noted

- Do Not Modify State Directly - Use setState()
    - For example, this will not re-render a component:
    - this.state.comment = 'Hello';
    - State Updates May Be Asynchronous

- React may batch multiple setState() calls into a single update for performance.
- Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.
- A component may choose to pass its state down as props to its child components

Topic 16.24: State in Functional components

- Earlier, Functional Components were stateless.
- React Hooks made it possible to have state in Function Components.
- Hooks are a new addition in React 16.8.
- They let you use state without writing a class.
- Hooks allow you to reuse stateful logic without changing your component hierarchy.

Topic 16.25: Conditional Rendering

- **Conditional rendering** as a term describes the ability to render different UI markup based on certain conditions.
- Conditional rendering in React works the same way conditions work in JavaScript.
- Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.
    - If/else
    - element variables
    - Ternary operator
    - Short Circuit Evaluation with &&

Topic 16.26: State in Functional components

- Earlier, Functional Components were stateless.
- React Hooks made it possible to have state in Function Components.
- Hooks are a new addition in React 16.8.
- Hooks are functions that let you “hook into” React state and lifecycle features from function components.

Topic 16.27: Benefits

- They let you use state without writing a class.
- It’s hard to reuse stateful logic between components
- Hooks allow you to reuse stateful logic without changing your component hierarchy
- Complex components become hard to understand.
- In many cases it’s not possible to break these components into smaller ones because the stateful logic is all over the place.

Topic 16.28: Example

- import React, { useState } from 'react';
- function Example() {

```
// Declare a new state variable, "count"
const [count, setCount] = useState(0);
return (
  <div>
    <p>You clicked {count} times</p>
    <button onClick={() => setCount(count + 1)}>
      Click me
    </button>
  </div>
);
}
```

Topic 16.29: Hooks

- React provides a few built-in Hooks like useState.
- You can also create your own Hooks to reuse stateful behavior between different components.
- Rules of Hooks
- Hooks are JavaScript functions, but they impose two additional rules
- Only call Hooks at the top level. Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks from React function components. Don't call Hooks from regular JavaScript functions.

Topic 16.30: Built in Hook

- useState
- const [count, setCount] = useState(0);
- useEffect
- By using this Hook, you tell React that your component needs to do something after render.
- useEffect Hook is componentDidMount, componentDidUpdate, and componentWillUnmount combined.

Topic 16.31: Context

- In a typical React application, data is passed top-down (parent to child) via props. When you had to pass props several components down your component tree. It results in prop drilling.

Topic 16.32: React Context

- There are two use cases when to use it:
- When your React component hierarchy grows vertically in size and you want to be able to pass props to child components without bothering components in between.
- When you want to have advanced state management in React. Doing it via React Context allows you to create a shared and global state.

17. CS 15 REACT Continued

Topic 17.1 State

- Components need to “remember” things
- React Class components have a built-in state object.
- You can add state to a component with a useState Hook.
- The state object is where you store property values that belongs to the component.
- When the state object changes, the component re-renders.

Topic 17.2 State in Functional components

- Earlier, Functional Components were stateless.
- React Hooks made it possible to have state in Function Components.
- Hooks are a new addition in React 16.8.

Topic 17.3 Example

```
import React, { useState } from 'react';
function Example() {
 // Declare a new state variable, "count"
 const [count, setCount] = useState(0);
 return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={() => setCount(count + 1)}>
    Click me
   </button>
  </div>
 );
}
```

Topic 17.4 Points to be Noted

- Do Not Modify State Directly - Use setState()
  - For example, this will not re-render a component:
  - this.state.comment = 'Hello';
  - If two state variables always update together, consider merging them into one.
  - React may batch multiple setState() calls into a single update for performance.
- A component may choose to pass its state down as props to its child components

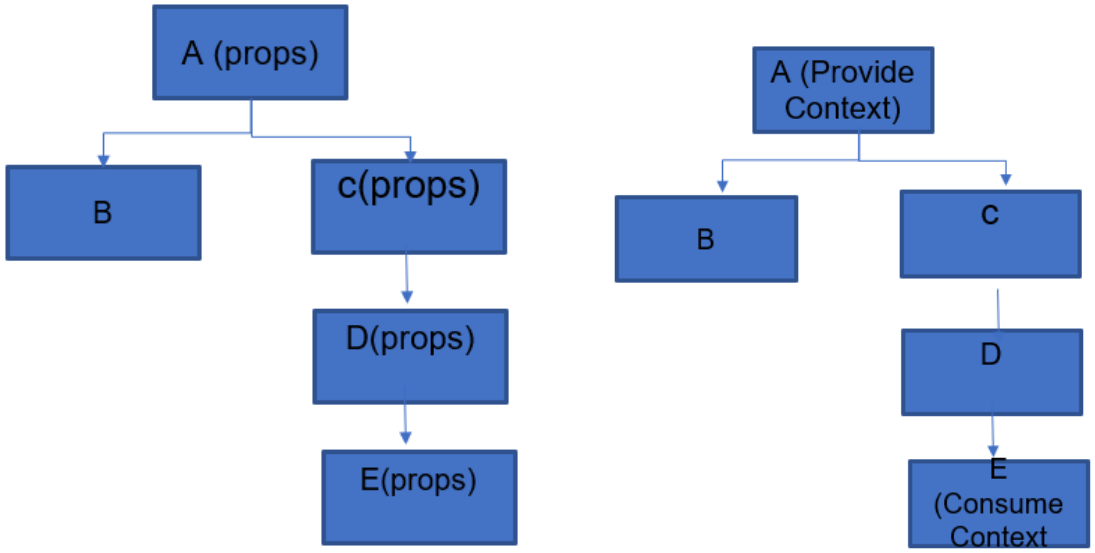
Topic 17.5 Hooks

- React provides a few built-in Hooks like useState.

- You can also create your own Hooks to reuse stateful behavior between different components.
- Rules of Hooks
- Hooks are JavaScript functions, but they impose two additional rules
- Only call Hooks at the top level. Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks from React function components. Don't call Hooks from regular JavaScript functions.

Topic 17.6 Context

- In a typical React application, data is passed top-down (parent to child) via props. When you had to pass props several components down your component tree. It results in prop drilling.



Topic 17.7 React Context

- There are two use cases when to use it:
- When your React component hierarchy grows vertically in size and you want to be able to pass props to child components without bothering components in between.
- When you want to have advanced state management in React. Doing it via React Context allows you to create a shared and global state.

Topic 17.8 Conditional Rendering

- **Conditional rendering** as a term describes the ability to render different UI markup based on certain conditions.
- Conditional rendering in React works the same way conditions work in JavaScript.
- Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.
- If/else
- element variables

- Ternary operator
- Short Circuit Evaluation with &&

Topic 17.9 Responding to events

- React lets you add *event handlers* to your JSX.
- Built-in components like <button> only support built-in browser events like onClick.

Topic 17.10 React Router- Routing

- SPAs dynamically update sections of a page without full-page reloads
- SPAs introduce challenges, such as managing browser history and routing.

Topic 17.11 React Router

- React Router is the most popular routing library for React
- Enables navigation among views
- Handle routing *declaratively*.
- <Route path="/home" component={Home} />
- React Router includes three main packages:
    - react-router: This is the core package for the router
    - react-router-dom: It contains the DOM bindings for React Router. In other words, the router components for websites
    - To get started:
    - npm install — save react-router-dom

Topic 17.12 Using React Router

- The React Router API is based on three components:
- Router: At the core of React Router v6 is the Router component, which provides routing capabilities to the application.
- Routes: The Routes component is used to define route configurations within the application. It replaces the previous Switch component and allows for more flexible route matching and rendering. Routes can be nested and include route elements defined with the Route component.
- The Route component defines a route within the application and specifies the component to render when that route matches the current URL. Routes in React Router v6 use an element prop instead of component, and they match paths inclusively by default.

Topic 17.13 <Router>

- React Router v6 offers a single <Router> component that abstracts away the underlying routing strategy
- The main job of a <Router> component is to create a history object to keep track of the location (URL).
- When the location changes because of a navigation action, the child component is re-rendered.
- The react-router-dom package offers three higher-level, ready-to-use router components, as

- BrowserRouter: The BrowserRouter component handles routing by storing the routing context in the browser URL and implements backward/forward navigation with the inbuilt history stack
- HashRouter: Unlike BrowserRouter, the HashRouter component doesn't send the current URL to the server by storing the routing context in the location hash (i.e.,i ndex.html#/profile)
- MemoryRouter: This is an invisible router implementation that doesn't connect to an external location, such as the URL path or URL hash. The MemoryRouter stores the routing stack in memory but handles routing features like any other router

Topic 17.14 <Route>

- It renders some UI if the current location matches the route's path.
- <Route path="/about" component={About}/>
- By default, routes are inclusive, more than one <Route> component can match the URL path and render at the same time.

Topic 17.15 <Link>

- Create a navigational link to navigate to particular URL
-
- <Link to="/">Home</Link>
- <Link to="/about">About</Link>
-

Topic 17.16 <Outlet>

- An <Outlet> should be used in parent route elements to render their child route elements.
- This allows nested UI to show up when child routes are rendered.
- If the parent route matched exactly, it will render a child index route or nothing if there is no index route.

Topic 17.17 Redux

- Redux is a predictable state container for JavaScript apps.
- Used for application State Management
- It is inspired by Facebook's Flux Architecture

Topic 17.18 Need for Redux

- In React, data flows from one parent component to the child component.
- The Data flow is unidirectional
- The child components cannot pass data to parents
- The non-parents components cant communicate with each other
- Redux offers a Store- to store all your application state in one place-single source of truth
- Components can dispatch the state changes to the Store- instead of sending it to each components
- Components need the data can subscribe to the State

Topic 17.19 Principles of Redux

- A single object tree stores all of your application state
- State is read only and changes are triggered by actions.
- The state can only be manipulated by pure functions that are triggered by your actions

Topic 17.20 Actions

- An action is simply an object that describes a change you want to make to your state.
- Actions are payload on information that send data from your application to the store.
- A standard pattern for actions is the following structure:
    - const action = {
    - type: 'ACTION_TYPE',
    - payload: 'Some data'
    - };
    - The payload is the data that will be used to transform our state.
    - const increment = {
    - type: 'INCREMENT'
    - };
    - Some actions like incrementing a number do not require any additional data

Topic 17.21 Action Creator

- Function which create actions
- export const addNumber = (number) => ({
- type: ADD_NUMBER,
- payload: number
- });
- const action = addNumber(7);
- Action creators are simply a function that allow us to abstract away the creation of actions, allowing us to easily dispatch an action without having to define all of its properties.

Topic 17.22 Reducers

- A reducer is the pure function that we will use to transform our store state.
- Actions describe the data that needs to be change, But How the state change happens is the responsibility of the Reducer
- Reducers are triggered whenever an action is dispatched and receive both the current state of that reducer (which will be undefined to begin with) and the action that was dispatched.
- export const count = (state = 0, action) => {
- switch (action.type) {
- case ADD_NUMBER:
- return state + action.payload;

- default:
- return state;
- }
- };

Topic 17.23 Reducer

- Takes in the prevstate and action, and determines what sort of update needs to be done based on action type
- It returns the newstate value
- It returns the prevstate, If no change occurred
- Reducers are pure functions, they do not modify the passed original state, but make their own copy and updates them
- Single Store , Multiple reducer
- Root reducer slices up the state based on keys

Topic 17.24 Store

- The Store is the object that hold application state.
- Create a Store
- export const store = createStore(count);
- store.dispatch(action) is the method that is used to dispatch an action and subsequently trigger our reducers.
- store.getState() simply returns the current state of the store.
- Register listeners via the subscribe() method
- Store Calculates the state changes and communicate to the Root reducer

Topic 17.25 Combining Reducers

- For most applications we are going to want to store more than a single number
- export const store = createStore(combineReducers({
- count,
- someOtherReducer
- }));
- Behind the scenes is creating another function that calls all our our reducers with the state that is relevant to them.

18. CS 16 Accessibility and Performance

Topic 18.1: UX based on Technologies used

- Consider the technologies that people use to experience our websites
    - devices,
    - browsers,
    - operating systems, and
    - assistive technologies
- How to design for them
- **Responsive design** is about creating fluid designs through HTML and CSS that adapt seamlessly to screen size, resolution, and aspect ratio, so that the layout remains optimally usable and attractive
- **Accessible design** is about creating designs that are usable and enjoyable by people with disabilities, including physical, sensory, cognitive, and neurological problems
- **Universal design** is about creating designs that are usable and enjoyable by everyone, regardless of age, status, culture, ethnicity, or ability

Topic 18.2: Accessibility

- Consider the technologies that people use to experience our websites
    - devices,
    - browsers,
    - operating systems, and
    - assistive technologies
- Accessible design is about creating designs that are usable and enjoyable for people with disabilities.

Topic 18.3: Designing for accessibility

- The most important way to design for assistive technologies is to design with different users in mind, following usability guidelines
- Designing for keyboard input
- Designing for Screen Readers
- Careful color and color contrast choices
- Web accessibility is about removing barriers that prevent people with disabilities from accessing websites.

Topic 18.4: Principles of accessibility

    - The **Web Content Accessibility Guidelines *(*WCAG 2.0*)*** by W3C defines
    - Four principles of accessibility.
- **Perceivable**: Information and other interface elements must be visible to everyone.
- **Operable**: Everyone must be able to navigate around any website.

- **Understandable**: The information on websites must be easily understandable.
- **Robust**: The content and its interpretation must be reliable, and compatible with different devices, browsers, and assistive technologies.

Topic 18.5: Guidelines

- Using semantic HTML elements.
- Use Text Alternatives for non text content like images
    -
- Ensure that form elements are accessible.
- Arrange the HTML document with a logical and hierarchical structure.
- Use appropriate heading elements
- Ensure that all interactive elements are accessible via the keyboard.
- Maintain sufficient contrast between text and background colors to enhance readability.

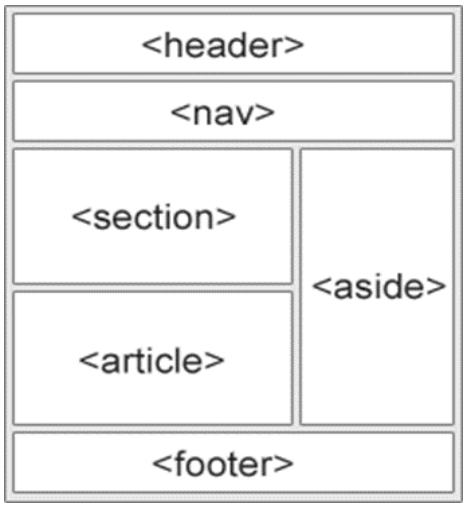
Topic 18.6: An army of elements

```
<div class="container">
    <div class="row ">
        <div class="col-2 bg-primary">first column</div>
        <div class="col-6 bg-secondary">second column</div>
        <div class="col-4 bg-primary">third column</div>
    </div>
</div>
<hr/>
<div class="container">
    <div class="row ">
        <div class="col-sm-2 bg-primary">first column</div>
        <div class="col-sm-6 bg-secondary">second column</div>
        <div class="col-sm-4 bg-primary">third column</div>
    </div>
    <hr/>
    <div class="row ">
        <div class="col-md-2 bg-primary">first column</div>
        <div class="col-md-6 bg-secondary">second column</div>
        <div class="col-md-4 bg-primary">third column</div>
    </div>
```

Topic 18.7: Semantic Elements

- header: <header>.
- navigation bar: <nav>.
- main content: <main>,
- Main content can be organized into subsections by

    - <article>, and
    - <section>
- sidebar: <aside>; often placed inside <main>.
- footer: <footer>.



Topic 18.8: ARIA

- The Web Accessibility Initiative's Accessible Rich Internet Applications specification (WAI-ARIA)
- ARIA provides additional information to assistive technologies about the roles, properties and states of elements.

Example

```
<button aria-expanded="false" onclick="toggleCollapsible()">
  Toggle Section
</button>
<div class="content" aria-hidden="true">
   <p>This is the content of the collapsible section.</p>
</div>
```

Topic 18.9: Testing for accessibility

- **Keyboard only**: Check if you can access all parts of your website while only using the keyboard
- **Screen reader**: Download a screen reader or use the built in one for your OS and view your website through it.
- **Magnification**: Enlarge your font to 200%.
- **Checklists**: There are many accessibility checklists available to check your designs and HTML against, based on the WCAG guidelines.
    - WebAIM (http://webaim.org/standards/wcag/checklist/)
- **WAVE**: WebAIM also provides the WAVE extension for Chrome and Firefox, which checks accessibility automatically

Topic 18.10: Self Reading

- https://testsigma.com/tools/accessibility-testing-tools/

19. CS 16 Testing

Topic 19.1: **Frontend Testing**

Topic 19.2: Test Pyramid

- But when it comes to automated tests, how many of each test do we want?

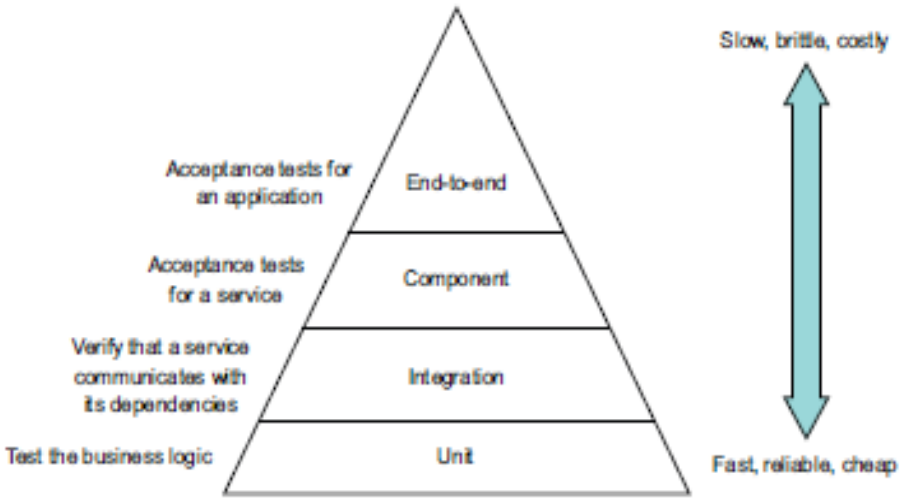


Figure 9.5 The test pyramid describes the relative proportions of each type of test that you need to write. As you move up the pyramid, you should write fewer and fewer tests.

Topic 19.3: Frontend Testing

- Unit Testing
  - Developers isolate the minor application components, check their behavior, and identify defects early in the development pipeline.
- Visual Regression Testing
  - Also sometimes called visual snapshot tests
  - Comparing screenshots taken before and after code changes
- Cross Browser Testing
  - check if a website works as expected when accessed via different browser-device-OS combinations.
  - Responsive Design
- Accessibility Testing
  - checks if every user on the internet can easily access a website, including individuals with special needs
- Acceptance Testing

Topic 19.4: Jest

- Jest comes with a test runner, assertion library, and good mocking support
- Test runner — a tool that picks up files that contain unit tests, executes them, and writes the test results to the console or log files. It automatically finds tests to be executed in your repository
- Assertion library — verifies the results of a test.
- Mocks — used in unit testing a component. A component under test has many dependencies. Jest automatically mocks the dependencies when you're running your tests.
- Mocking library — facilitates the usage of mocks in unit testing.

Topic 19.5: Jest Test Suite

- describe('my function or component', () => {
- test('does the following', () => {
- expect(true).toBe(true);
- });
- });
- describe-block is the test suite,
- the test-block (which also can be named it instead of test) is the test case
- Expect() is the assesrtion

Topic 19.6: The React Testing Library

- The React Testing Library is a very light-weight solution for testing React components.
- It provides light utility functions on top of react-dom and react-dom/test-utils
- The utilities this library provides facilitate querying the DOM in the same way the user would.
- Finding form elements by their label text (just like a user would), finding links and buttons by their text (like a user would).

Topic 19.7: API Testing

- Scope is one way of characterizing tests
- Unit tests—Test a small part of a service, such as a class.
- Integration tests—Verify that a service can interact with infrastructure services such as databases and other application services.
- Component tests—Acceptance tests for an individual service.
- End-to-end tests—Acceptance tests for the entire application.

Topic 19.8: Unit Testing

- These are tests that typically test a single function or method call.
- These are tests that help the developers and so they would be technology-facing, not business facing
- The prime goal of these tests is to give us very fast feedback about whether our functionality is good.

Topic 19.9: Types of Unit Testing

- Solitary unit test—Tests a class in isolation using mock objects for the class's dependencies
- Sociable unit test—Tests a class with its dependencies

Topic 19.10: Integration Tests

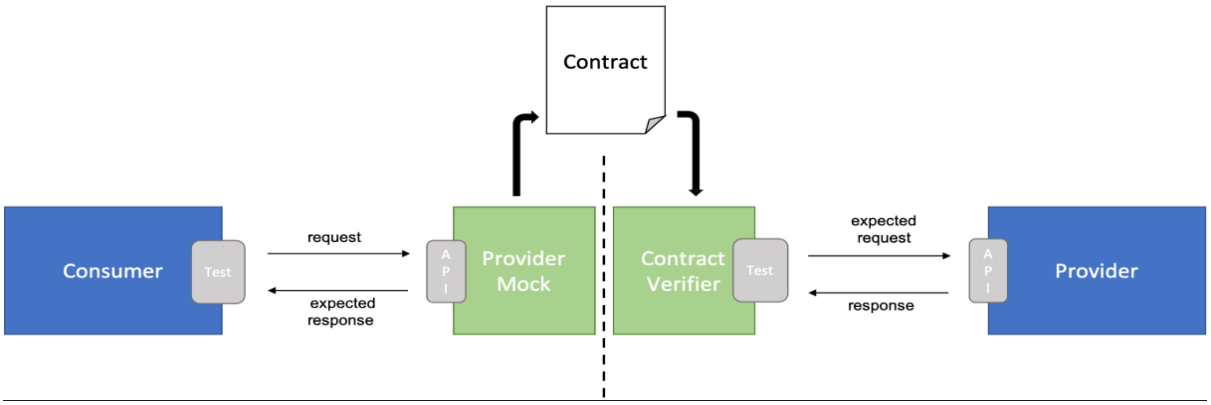
- They verify that a service can properly interact with infrastructure and other services.
- Postman can be used effectively for integration testing, where you test how different components of your application interact.

Topic 19.11: Contract testing

- Contract testing is a software testing methodology that tests the interactions between different microservices or software components based on their contracts.
- In contract testing, each service or component is given a contract, which defines how to work with the service and which responses to accept.
- There are two types of contract testing: consumer-driven and provider-driven.

- **Consumer-Driven Contract Testing**
- Consumer-driven contract testing is a contract testing approach in which the service consumer, for example, a client application, defines the contracts the service provider must adhere to.
- **Provider-Driven Contract Testing**
- In provider-driven contract testing, the service provider defines the contracts that the consumer (i.e., the client) must adhere to when consuming the service. These contracts are usually defined in a format that can be shared between the service provider and the client, like OpenAPI or Swagger.

Topic 19.12: Consumer-driven Contract Testing



Topic 19.13: Self Reading

- https://semaphoreci.com/blog/test-microservices