Birla Institute of Technology & Science, Pilani
Work Integrated Learning Programmes Division
Second Semester 2023-2024
Solution Key
Mid-Semester Examination
(EC-2 Regular)

Course No.	:	SESAPZG503
Course Title	:	Full Stack Application Development
Nature of Exam	:	Closed Book
Weightage	:	30%
Duration	:	1 Hour 30 Min
Date of Exam	:	06-04-2024 (FN)

No. of Pages = 2
No. of Questions = 4

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. Explain each of the following concepts of Client-Server architecture with appropriate examples. **[4 Marks]**

(a) Asymmetrical Protocols
(b) Shared Resources
(c) Encapsulation of Services
(d) Extensible design

Explanation 0.5 marks, Example 0.5 marks

(a) In client-server communication, protocols define how data is exchanged. Asymmetrical protocols assign different roles and capabilities to clients and servers. There is a many-to-one relationship between clients and a server. Clients always initiate a dialog by requesting a service. Servers wait passively for requests from clients. The client is active and the server is passive, so it is asymmetrical.

A common example of an asymmetrical protocol is the Hypertext Transfer Protocol (HTTP) used for web communication. In this protocol, the client (such as a web browser) initiates requests for resources (e.g., web pages) from the server, which then responds to these requests. The client is responsible for sending requests, while the server is responsible for processing these requests and sending back the requested data. The roles of the client and server are clearly defined and cannot be interchanged.

(b) Shared resources in client-server architecture refer to resources such as files, databases, or hardware devices that are accessible and utilized by multiple clients or users through a central server. The server manages access to these resources, ensuring that they are shared efficiently and securely among clients.

A file server in a network environment is a typical example of shared resources in client-server architecture. Multiple clients connected to the network can access files stored on the file server. When a client needs to access a file, it sends a request to the server, which then grants access to the file and handles file operations such as reading or writing. The server ensures proper synchronization and access control to prevent conflicts and maintain data integrity.

(c) Encapsulation in client-server architecture refers to the packaging of server functionality as independent services. Clients interact with these services without needing to know the internal workings of the server.
An email server provides services like sending, receiving, and managing email messages. Clients (email applications) interact with these services through protocols like POP3 or IMAP, focusing on composing and sending emails without needing to manage the underlying storage or delivery mechanisms.

(d) Extensible design in client-server architecture refers to the ability of the system to accommodate changes and additions without requiring significant modifications to the existing architecture or core functionalities. An extensible design allows for easy integration of new features, services, or technologies, thereby enhancing the scalability and flexibility of the system.
An example of an extensible design in client-server architecture is a web server that supports plugins or extensions. These plugins can add new functionalities such as authentication methods, caching mechanisms, or support for additional protocols. The server architecture is designed to accommodate these extensions seamlessly, allowing administrators to enhance the server's capabilities without disrupting its existing functionalities. This extensibility enables the server to adapt to evolving requirements and technologies effectively.

Q.2. Compare and contrast the usage of **HTTP Polling**, **Server Sent Events** (SSE), and **WebSockets** for real-time communication in web applications, considering their trade-offs. **[4 Marks]**

Feature	HTTP Polling	Server Sent Events (SSE)	WebSockets
Protocol	Uses HTTP	Uses HTTP	Uses WebSocket protocol
Connection	Client repeatedly polls the server	Server pushes events to the client	Full-duplex communication channel
Real-Time	Not real-time, latency depends on polling interval	Real-time, immediate updates	Real-time, immediate updates
Efficiency	Inefficient due to constant polling	More efficient than polling	Highly efficient and low latency
Server Load	Higher server load due to frequent requests	Lower server load as events are pushed	Moderate server load
Compatibility	Compatible with most browsers	Compatible with modern browsers	Requires modern browser support
Error Handling	May miss real-time updates if polling interval is too long	Reliable real-time updates	Reliable real-time updates
Use Cases	Suitable for scenarios where real-time updates are not critical	Suitable for applications requiring real-time updates, such as stock tickers or chat applications	Suitable for highly interactive applications like online gaming or collaborative editing

(b)Although HTTP polling is not that efficient as other real time communication methods, describe **four scenarios** with example where this can be useful.

Refer Page 58 – Class Discussion3

1. Social Media Feeds
2. Stock Market Updates
3. Weather Updates
4. Online Gaming

Q.3. Answer following questions: **[4+2+2+4 = 12 Marks]**

(a) Explain with a diagram the **process** and **challenges** involved in transforming an HTTP/1 message into an HTTP/2 message.
(b) Explain with example key differences between the two protocols.
(c) Are there cases where HTTP/1 is preferred over HTTP/2 (provide examples) ?
(d) Convert following HTTP/1 message to HTTP/2

```
POST /login HTTP/1.1
Host: bits-pilani.ac.in
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/97.0.4692.71 Safari/537.36
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 48
Connection: keep-alive

{"username": "student", "password": "welcome"}
```

(a) You must answer with Diagram similar to Page 39,41,42 **Class Discussion-3** materials.

Process:

1. **Parsing:** The HTTP/1 message is parsed to extract its components like method, URL, headers, and body.
2. **Header Conversion:** Headers are converted to a format suitable for HTTP/2. This involves:
    - **HPACK Encoding:** Redundant header information across requests is identified and compressed using the HPACK algorithm, reducing overall message size.
    - **Pseudo-headers:** Special headers specific to HTTP/2 (like ":authority") are added.
3. **Stream Mapping:** The message is divided into one or more streams based on its purpose (e.g., separate streams for HTML content, images, etc.).
4. **Framing:** The data (headers and body) is segmented and encapsulated into HTTP/2 frames. Each frame has a header indicating its type, length, and other details.
5. **Transmission:** Frames are sent over a single TCP connection in any order, allowing for multiplexing.

Challenges:

- **Legacy Applications:** Converting existing HTTP/1 applications might require code modifications to leverage HTTP/2 features effectively.
- **Server Configuration:** Servers need to be configured to support HTTP/2 features like multiplexing and HPACK encoding.
- **Intermediaries:** Firewalls, proxies, or load balancers may need adjustments to handle the binary framing and multiplexing nature of HTTP/2.

Feature	HTTP/1.1	HTTP/2
Multiplexing	Single connection per request	Single connection with multiplexing
Headers	Text-based, uncompressed	Binary, HPACK compressed for redundancy reduction
Request Order	Blocking (one request at a time)	Non-blocking (multiple requests concurrently)
Server Push	Not supported	Supported (server can proactively send resources)

(d) **Diagram** Refer Page 45 of Class Discussion-3

HTTP/1	**HTTP/2**
POST /login HTTP/1.1 Host: bits-pilani.ac.in User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/97.0.4692.71 Safari/537.36 Accept: */* Accept-Language: en-US,en;q=0.5 Accept-Encoding: gzip, deflate Content-Type: application/json Content-Length: 48 Connection: keep-alive {"username": "student", "password": "welcome"}	POST /login **HTTP/2** :**method**: POST :**scheme**: https :**path**: /login :**host**: bits-pilani.ac.in :**user-agent**: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/97.0.4692.71 Safari/537.36 :authority: bits-pilani.ac.in :accept: */* :accept-language: en-US,en;q=0.5 :accept-encoding: gzip, deflate :content-type: application/json :content-length: 48 :connection: keep-alive {"username": "student", "password": "welcome"}

(c)
Yes, there are still some cases where HTTP/1 might be preferred over HTTP/2:

- **Legacy Systems Compatibility**: Some older systems or devices might not support HTTP/2, or they might have limited support. In such cases, using HTTP/1 might be necessary to ensure compatibility.
- **Resource Constraints**: HTTP/2, being more complex than HTTP/1, can consume more server resources, especially in terms of CPU and memory. In resource-constrained

environments, such as embedded systems or very low-powered devices, HTTP/1 might be preferred to minimize resource usage.

- **Debugging and Troubleshooting**: Debugging HTTP/2 traffic can be more complex than HTTP/1 due to its binary framing layer. In scenarios where detailed debugging or troubleshooting is required, HTTP/1 might be preferred for its simpler textual representation.
- **Latency Sensitive Applications**: In some latency-sensitive applications, the additional overhead introduced by HTTP/2, such as header compression and stream multiplexing, might outweigh its performance benefits. HTTP/1's simpler request-response model might be preferred in such cases.
- **Server Push Not Needed**: If a web application doesn't benefit from server push capabilities provided by HTTP/2, sticking with HTTP/1 might be more straightforward and efficient.

Q.4. You're tasked with designing an API for an e-commerce platform specializing in electronic gadgets. Your objective is to create a robust system that enables users to browse products, add them to their cart, and place orders. Develop RESTful endpoints and GraphQL schema & queries based on the following requirements:

[2+1+1+3+3 = 10 Marks]

(a) Product Catalog Management: Design REST endpoints to allow users to:
   i. Retrieve a list of all available products and View details of a specific product by its ID.
   ii. Search for products based on various criteria such as category, brand, or price range.
   iii. Add a new product to the catalog and (for admin users only), Update the details of an existing product (for admin users only) and
   iv. Delete a product from the catalog (for admin users only).

(b) User Cart Management: Implement REST endpoints for managing user carts, including:
   i. Adding a product to the user's cart.
   ii. Viewing the contents of the user's cart.

(c) Order Processing: Create RESTful endpoints to facilitate order processing, including:
   i. Retrieving a user's order history.
   ii. Cancelling an existing order.

(d) Design a GraphQL schema for representing the system functionalities.

(e) GraphQL Queries: Utilize GraphQL to define queries for the following tasks:
   i. Fetch details of a specific product including its availability and price.
   ii. Retrieve a user's cart contents along with product details.
   iii. Get information about a specific order including the list of products and their quantities.

RESTful Endpoints:

Product Catalog Management:

GET /api/products: Retrieves a list of all available products.
GET /api/products/{productId}: Retrieves details of a specific product by its ID.
GET /api/products/search: Allows searching for products based on criteria such as category, brand, or price range. **or**

GET
/products?category={category}&brand={brand}&minPrice={minPrice}&maxPrice={maxPrice}

POST /api/products (requires admin authentication) Adds a new product to the catalog.
PUT /api/products/{productId} (requires admin authentication) (Admin only): Updates the details of an existing product.
DELETE /api/products/{productId} (Admin only): Deletes a product from the catalog.

User Cart Management:

POST /api/cart/add: Adds a product to the user's cart.

Or

POST /api/cart/items
Request Body: { "productId": "{productId}", "quantity": {quantity} }

GET /api/cart: Retrieves the contents of the user's cart.
DELETE /api/cart/{productId}: Removes a product from the user's cart.
DELETE /api/cart/clear: Clears the entire cart.

Order Processing:

POST /api/orders: Places a new order.
GET /api/orders: Retrieves the user's order history.
DELETE /api/orders/{orderId}: Cancels an existing order.
GET /api/orders/{orderId}/status: Tracks the status of an order.

==========
GraphQL Queries:

```
type Product {
  id: ID!
  name: String!
  description: String!
  price: Float!
  availableQuantity: Int!
}

type User {
  id: ID!
  name: String!
  email: String!
  cart: [CartItem!]!
  orders: [Order!]!
}

type CartItem {
  id: ID!
  product: Product!
  quantity: Int!
}

type Order {
```

```
  id: ID!
  status: String!
  products: [OrderedProduct!]!
}

type OrderedProduct {
  product: Product!
  quantity: Int!
}

type Query {
  products: [Product!]!
  product(id: ID!): Product
  searchProducts(category: String, brand: String, minPrice: Float, maxPrice: Float):
[Product!]!
  user: User
  order(id: ID!): Order
}

type Mutation {
  addProduct(name: String!, description: String!, price: Float!, availableQuantity: Int!):
Product!
  updateProduct(id: ID!, name: String, description: String, price: Float, availableQuantity:
Int): Product
  deleteProduct(id: ID!): ID
  addToCart(productId: ID!, quantity: Int!): CartItem!
  removeFromCart(productId: ID!): ID
  clearCart: ID
  placeOrder: Order!
  cancelOrder(orderId: ID!): ID
}
```

==========

Product Details:

```
query ProductDetails($productId: ID!) {
  product(id: $productId) {
    id
    name
    description
    price
    availableQuantity
  }
}
```

User Cart Contents:

```
query UserCartContents {
  user {
    id
    cart {
      id
      product {
        id
```

```
      name
      price
    }
    quantity
  }
 }
}
```

Order Information:

```
query OrderInformation($orderId: ID!) {
 order(id: $orderId) {
  id
  status
  products {
   id
   name
   price
   quantity
  }
 }
}
```