Web Applications and Services

Question 1

(a) Are the following statements true or false? Briefly explain your answer.

i) There is only one type of HTTP connection.

Ans: False. There are two types of HTTP connections: persistent connections and non-persistent connections.

Persistent connections allow multiple HTTP requests and responses to be sent over a single TCP connection. This helps reduce the overhead of establishing and tearing down TCP connections for each request, improving performance.

Non-persistent connections, on the other hand, establish a new TCP connection for each HTTP request and response. This can lead to increased overhead and slower performance compared to persistent connections.

The choice between persistent and non-persistent connections depends on various factors, such as the nature of the application, the expected traffic volume, and the specific requirements of the system.

ii) HTTP response messages never have an empty message body.

Ans: False. HTTP response messages can have an empty message body.

In some cases, a response may not need to include a message body, especially for certain status codes like 204 No Content or 304 Not Modified. These status codes indicate that the server successfully processed the request, but there is no new content to send back to the client. In such cases, the response headers are still included, but the response message body is empty.

Additionally, even for other status codes where a response message body is typically included, there may be situations where the server decides not to include a message body. This could occur, for example, when the server responds with a redirection (3xx status code) and wants to indicate the new location to the client without providing any additional content in the message body.

So, while it is common for HTTP responses to include a message body, it is not mandatory, and there are cases where an HTTP response can have an empty message body.

iii) Cookies, in HTTP messages, are the only way to keep state.

Ans: False. While cookies are a commonly used method for maintaining state in HTTP messages, they are not the only way to do so. There are other mechanisms available for maintaining state in HTTP, including:

1. Session IDs: Session IDs are often used to associate a user's state with their session on the server. The session ID is typically stored on the server, and a session cookie containing the session ID is sent to the client to maintain the association between the client and server session.
2. URL Parameters: State information can be passed through URL parameters. The server can include specific parameters in the URLs it generates, and the client can include those parameters in

subsequent requests to maintain state. This approach is often used in web applications that require bookmarkable URLs or in cases where cookies are disabled or not supported.

3. Hidden Form Fields: State information can be stored in hidden form fields in HTML forms. When the form is submitted, the state information is included in the request payload, allowing the server to maintain state between requests.
4. HTTP Headers: Custom HTTP headers can be used to store and transmit state information. This approach is less common and typically used in specific cases where other mechanisms are not suitable.

iv) A web cache acts mainly as a server, satisfying client requests without involving the origin server.

Ans: True. A web cache acts as an intermediary between the client and the origin server. Its primary purpose is to store copies of web content (such as HTML pages, images, and other resources) and serve those copies to clients without involving the origin server.

When a client requests a resource, the web cache checks if it has a cached copy of the resource. If the cache has a valid copy, it can directly serve that copy to the client without contacting the origin server. This process is known as a cache hit. It helps to improve response times and reduce network traffic by serving content from a nearby cache instead of making a round trip to the origin server.

v) In HTTP v1.1, a server responds in FCFS order to GET requests. However, this may cause problems.

Ans: False. In HTTP/1.1, servers do not respond to GET requests in a First-Come-First-Serve (FCFS) order. Instead, the server responds to requests based on the order it receives the TCP packets containing those requests.

(b) Explain how third-party cookies enable tracking a user's browsing behaviour

Ans: Third-party cookies enable tracking a user's browsing behavior by allowing external entities, such as advertisers or analytics providers, to store and retrieve information about the user across different websites. Here's a general explanation of how this tracking mechanism works:

1. Origin Website: When a user visits a website, let's call it the "origin website," it may include resources (such as images, scripts, or ads) from other domains or "third-party domains." These third-party domains are different from the domain of the origin website.
2. Setting Third-Party Cookies: If the third-party domain has the ability to set cookies, it can include a cookie in the HTTP response headers when its resources are loaded on the user's browser. This cookie is typically stored on the user's device and associated with the third-party domain.
3. Cross-Site Requests: As the user navigates to other websites, the user's browser may automatically send requests to the third-party domain if the origin website included resources from that domain. These requests include the stored third-party cookies in the HTTP headers.
4. Tracking User Activity: When the third-party domain receives these cross-site requests with its cookies, it can track the user's browsing behavior. By analyzing the requests and associated cookies from various websites, the third-party domain can build a profile of the user's interests, behaviors, and preferences.
5. Data Aggregation: Multiple third-party domains may participate in tracking the user's browsing behavior. They can share and aggregate the collected data to create comprehensive user profiles.

This information is valuable for targeted advertising, personalized content delivery, or analytics purposes.

(c) Explain how HTTP v2 mitigates the head of line blocking. Give an example to justify your answer.

Ans: HTTP/2 addresses the issue of head-of-line blocking, which refers to the situation where the transmission of one resource is delayed due to the presence of another resource ahead of it in the transmission queue. HTTP/2 introduces several features to mitigate this problem:

1. Multiplexing: HTTP/2 introduces multiplexing, which allows multiple requests and responses to be interleaved and sent over a single TCP connection. This means that multiple resources can be requested and delivered concurrently without waiting for the completion of previous resources. This effectively reduces head-of-line blocking by allowing independent resources to be processed and transmitted in parallel.
2. Stream Prioritization: HTTP/2 introduces stream prioritization, where each request/response is assigned a priority value. This allows the server and client to determine the relative importance of different resources. The prioritization information is used to allocate resources and prioritize the transmission of critical resources ahead of less important ones. By managing the order of transmission based on priority, head-of-line blocking can be further minimized.

Example:

Let's consider a scenario where a webpage contains multiple resources, such as HTML, CSS, JavaScript, and images. In HTTP/1.1, these resources would typically be requested sequentially, which could lead to head-of-line blocking.

However, with HTTP/2, the browser can send multiple requests for different resources concurrently over a single TCP connection. For instance, it can request the HTML, CSS, and JavaScript files simultaneously. The server can respond to these requests and send the corresponding responses concurrently as well.

Even if the CSS response takes longer to generate on the server side, it does not block the delivery of the HTML response or any other resource. The browser can continue rendering and processing other resources as they arrive, eliminating the head-of-line blocking effect. This improves the overall page loading speed and user experience.

By allowing parallel processing and prioritization of resources, HTTP/2 significantly mitigates head-of-line blocking and improves the efficiency of web page loading.

(d) Explain why the intelligence is at the client when using Dynamic, Adaptive Streaming over HTTP (DASH).

Ans: In Dynamic Adaptive Streaming over HTTP (DASH), the intelligence is primarily located at the client-side. DASH is a video streaming protocol that dynamically adjusts the video quality and bitrate based on the client's network conditions and device capabilities. Here's why the client plays a crucial role in DASH:

1. Adaptive Bitrate Streaming (ABR): DASH employs ABR, which means the video is divided into smaller segments of different quality levels. The client dynamically selects the appropriate quality

level for each segment based on the current network conditions. The decision-making process is driven by client-side algorithms.

2. Network Adaptation: The client continuously monitors the network conditions, such as available bandwidth, latency, and packet loss. Based on this information, the client can adapt and switch to a higher or lower quality level to ensure smooth playback. It may also request different segments from different servers or CDNs to maximize performance.
3. Client-Side Buffering: The client buffers a certain amount of video content in advance to handle fluctuations in network conditions. It can make decisions on buffer management, such as adjusting the buffer size or prioritizing segment fetching, to optimize the playback experience and reduce interruptions.
4. Device-Specific Adaptation: The client is aware of the device capabilities, such as screen size, resolution, and decoding capabilities. It can tailor the video quality and format based on the device's capabilities to provide an optimal viewing experience.
5. Client-Side Quality Selection: The client can take into account user preferences and make decisions regarding video quality, such as allowing manual quality selection or applying predefined rules for quality adaptation.

By placing intelligence at the client, DASH enables adaptive streaming that is responsive to the client's network conditions and device capabilities. This approach allows for efficient utilization of network resources, improved video quality, reduced buffering, and a better overall streaming experience. Additionally, distributing the decision-making to the client side also reduces the server's computational load, making it scalable for large-scale streaming deployments.

Question 2

(a) Explain which components of the Django web framework implement parts of the MVC pattern. Give an example of such functionality for an e-commerce web application.

Ans: In the Django web framework, the components that implement parts of the MVC (Model-View-Controller) pattern are as follows:

1. Model: The Model component in Django represents the data structure and business logic of the application. It defines the data models that represent the database tables and relationships. The models handle the data retrieval, storage, and manipulation. The Django model component act as a Model (M) in the MVC pattern.

Example for an e-commerce web application: In an e-commerce application, the Model component in Django would define models such as Product, Order, User, etc. These models would have attributes and methods to handle data related to products, orders, users, and other relevant entities. For instance, the Product model could have fields like name, price, description, and methods to retrieve product information, update stock, etc.

2. View: The View component in Django handles the presentation logic and interacts with the user. It receives requests from the client, retrieves data from the model, and passes the data to the template for rendering. The view controls the flow of data between the model and the template. The view component of Django acts as a controller (C) in the MVC pattern.

Example for an e-commerce web application: In Django, a view function or class-based view would handle the logic for rendering product listings, order placement, user registration, etc. For instance, a view function for displaying a product detail page would retrieve the relevant product data from the model and pass it to a template for rendering. The view would handle any user interactions, such as adding the product to the cart or submitting a review.

3. Template: The Template component in Django defines the presentation and structure of the user interface. It is responsible for rendering the data received from the view and generating the HTML output that is sent to the client's browser. Lastly, the templates in Django act as the view (V) in the MVC pattern.

Example for an e-commerce web application: In Django, templates are typically written in HTML with embedded Django template tags and expressions. For example, a template for displaying a product detail page would define the layout and structure of the page, as well as placeholders for dynamic content. The template would use Django template tags and expressions to retrieve and display the product data received from the view.

(b) Describe the types of views available in Django's view layer. Justify your answer with examples, and listing all the advantages and disadvantages.

Ans: In Django's view layer, there are different types of views available to handle incoming requests and generate responses. Here are the main types of views in Django, along with their advantages and disadvantages:

1. Function-based Views: Function-based views are implemented as Python functions that accept an HTTP Request object as an argument and return an HTTP Response object.

- Example

```
def my_view(request):
    # view logic here
    return HttpResponse("Hello, World!")
```

- Advantages:
    - Simple and straightforward to implement.
    - Well-suited for basic views with minimal logic or for quick prototyping.
- Disadvantages:
    - Limited reusability as they cannot be easily shared or composed.
    - Difficult to organize and maintain for complex views with extensive logic or multiple HTTP methods.

2. Class-based Views: Class-based views are implemented as Python classes that inherit from Django's built-in view classes. They provide more structure and flexibility compared to function-based views.

- Example

```
from django.views import View
from django.http import HttpResponse

class MyView(View):
    def get(self, request):
        # view logic here
        return HttpResponse("Hello, World!")
```

- Advantages:
    - Encourage code reusability through class inheritance and mixins.
    - Provide pre-defined methods for different HTTP methods (e.g., get(), post(), etc.) for easy handling of different types of requests.
    - Support more complex view logic and allow for better organization and maintainability.
- Disadvantages:
    - Requires a deeper understanding of class-based views and inheritance.
    - Can be more verbose and require more initial setup compared to function-based views.

3. Generic Class-based Views: Django provides a set of pre-built generic class-based views that cover common use cases, such as creating, retrieving, updating, and deleting objects. These views are highly reusable and save development time.

- Example

```
from django.views.generic import ListView
from django.contrib.auth.models import User

class UserListView(ListView):
    model = User
    template_name = 'user_list.html'
```

- Advantages:
    - Highly reusable and cover common use cases out of the box.
    - Follow best practices and reduce code duplication.
    - Allow customization through subclassing and overriding methods.
- Disadvantages:
    - May require more configuration and understanding of class-based views.
    - Limited flexibility for complex requirements that deviate from the generic use cases.

4. API Views: API views are specialized views designed for building web APIs. They typically handle request/response serialization, authentication, and other API-specific functionalities.

- Example

```
from rest_framework.views import APIView
from rest_framework.response import Response

class MyAPIView(APIView):
    def get(self, request):
        # view logic here
        return Response("Hello, API!")
```

- Advantages:
    - Integrated with third-party libraries like Django REST Framework for building robust APIs.
    - Provide built-in support for serialization, authentication, and permissions.
- Disadvantages:
    - Targeted specifically for building APIs and may not be suitable for other types of views.
    - May introduce additional dependencies and complexity for simple view requirements.

The choice of view type depends on the specific needs of the project. Function-based views are suitable for simple or quick implementations, while class-based views and generic class-based views offer more flexibility and code organization. API views are ideal for building web APIs.

(c) Consider the sample URLconf below.

```
from django.urls import path
 from . import views
 urlpatterns = [
path('comments//', views.year_archive),
 ]
```

(i) Explain what it does, giving an example request and the Python function called.

Ans: The provided sample URLconf maps a URL pattern to a Python function in Django. Let's break down what it does:

- The URL pattern is defined as `'comments/<int:year>/'`. It starts with the literal string `'comments/'` and is followed by a variable portion `<int:year>`. The `<int:year>` part specifies that the variable should be an integer and assigns it the name `'year'`.

- The URL pattern is associated with the Python function `views.year_archive`. This means that when a request matches this URL pattern, the `year_archive` function will be called to handle the request.

Example request and Python function:

Suppose we have the Django application running, and a user makes the following request:

Request: `http://example.com/comments/2023/`

The requested URL matches the defined URL pattern `'comments/<int:year>/'`. The value `'2023'` is captured as the `year` variable.

The Python function called is `views.year_archive`. This function is responsible for handling the request and generating an appropriate response. It may look something like this:

~~~
```python code (view function)
def year_archive(request, year):
    # Perform logic based on the requested year
    # For example, retrieve comments for the given year from a database
    comments = Comment.objects.filter(year=year)

    # Generate the response, such as rendering a template with the comments
    return render(request, 'comments/year_archive.html', {'comments': comments})
```
~~~

In this example, the `year_archive` function takes two parameters: `request` and `year`. The `year` parameter corresponds to the captured value from the URL pattern.

The function can then perform any necessary logic based on the requested year, such as querying a database for comments made in that year. It can generate an appropriate response, such as rendering a template with the comments for the requested year.

(ii) Is it possible to restrict the user input only to valid years, i.e., with four digits? If necessary, justify your answer with Python code.

Yes, it is possible to restrict the user input to valid years with four digits in the provided URL pattern. One way to enforce this restriction is by using a regular expression in the URL pattern definition. Here's an example Python code demonstrating how it can be done:

~~~
```python code (Updated URL pattern)
from django.urls import path, re_path
from . import views

urlpatterns = [
    re_path(r'^comments/(?P<year>\d{4})/$', views.year_archive),
]
```
~~~

In this updated URLconf, the `re_path` function is used instead of `path` to support regular expressions. The regular expression pattern `r'^comments/(?P<year>\d{4})/$'` enforces that the captured `year` variable should be a sequence of exactly four digits.

The `^` character at the beginning and `$` at the end ensure that the regular expression matches the entire URL path, preventing partial matches. The `(?P<year>\d{4})` part captures the four-digit value and assigns it to the variable named `year`.

With this updated URL pattern, any URL that does not consist of four consecutive digits after the `'comments/'` portion will not match, effectively restricting the user input to valid four-digit years. For example, `'comments/2023/'` will match, while `'comments/23/'` or `'comments/20230/'` will not.

By using regular expressions in the URL pattern, you can enforce specific constraints on user input and ensure that only valid four-digit years are accepted in this case.

(d) Explain what view decorators are. Give a Python code example of its use to enable, for example, Django CSRF protection.

Ans: View decorators in Django are functions that wrap around a view function or class-based view to provide additional functionality or modify its behavior. They allow you to add functionality such as authentication, permission checks, caching, or enabling specific features to the views.

One common use case of view decorators in Django is to enable CSRF (Cross-Site Request Forgery) protection. CSRF is an attack that tricks users into performing unintended actions on a website without their knowledge or consent. Django provides a built-in decorator, csrf_protect, to protect against CSRF attacks.

Here's an example of using the csrf_protect decorator to enable CSRF protection on a view:

```
python
from django.views.decorators.csrf import csrf_protect
from django.http import HttpResponse

@csrf_protect
def my_view(request):
    # view logic here
    return HttpResponse("Hello, World!")
```

In the above example, the csrf_protect decorator is applied to the my_view function. This decorator adds CSRF protection to the view, ensuring that any form submissions made from the view require a valid CSRF token.

When a request is made to my_view, the csrf_protect decorator checks if the request is a POST request and if it contains a valid CSRF token. If the token is missing or invalid, Django raises a SuspiciousOperation exception. Django also provides other decorators for different purposes, such as login_required for enforcing authentication, permission_required for checking user permissions, and cache_page for caching views, among others.

(e) Consider the following statement “form interface elements can be simple and complex”. Is this statement true? Justify your answer giving some examples.

Ans: Yes, the statement "form interface elements can be simple and complex" is true. In web development, form interface elements can vary in complexity depending on the requirements and the data that needs to be captured from the user. Here are some examples to justify this statement:

1. Simple Form Elements:
    - Text Input: This is the most basic form element where users can enter a single line of text, such as their name or email address.
    - Checkbox: A simple checkbox allows users to select or deselect an option from a list of choices, such as agreeing to terms and conditions.
    - Radio Button: Radio buttons allow users to select a single option from a group of choices, such as selecting a gender or a preference.
    - Select Dropdown: A dropdown menu allows users to choose one option from a list, such as selecting a country or a category.
2. Complex Form Elements:
    - Text Area: A text area allows users to enter multiple lines of text, such as providing a detailed description or leaving comments.
    - File Upload: This element enables users to upload files from their local device, such as images, documents, or videos.
    - Date Picker: A date picker provides a user-friendly interface for selecting dates from a calendar, ensuring accurate date input.
    - Multi-Select Dropdown: This element allows users to select multiple options from a list, such as selecting multiple categories or tags.

Furthermore, form elements can become even more complex when combined with additional features and validations, such as:

- Form Validation: Form elements can have validation rules to ensure that the entered data meets specific criteria, such as required fields, valid email addresses, or password strength requirements.
- Conditional Fields: Forms can have fields that are dynamically shown or hidden based on the user's selections, making the form interface more adaptable and user-friendly.
- Complex Form Layouts: Forms can have more complex layouts involving grids, tabs, or accordion-style sections, providing a structured and organized user interface.

Overall, the simplicity or complexity of form interface elements depends on the specific requirements of the form and the data that needs to be captured. Simple elements are often used for basic data input, while complex elements and features are employed when more sophisticated interactions, validations, or data types are involved.