Q1. **API Definition**

The term API directly translates to Application Programming Interface. From my understanding, it is an interface that allows for communication between two different applications or components. The most commonplace relationship, at least in web development, is the communication between a client and a database. The client may send a GET request to the API which will communicate the type of data needed to the database, and then will format that data however the developer intends the consumer to use that data and return it to the client. It is a common misconception that the API is the data source when in reality it is the liaison from the client to the data source and the formatter for that data.

Q2. **gRPC exchange message format**

The format employed to exchange messages between machines using gRPC is called a Protocol Buffer (also known as a Protobuf). Protobuf is a very bare-bones and compact schema for quick transfer of data through the serialization of the messages and responses. A Protobuf is defined in a .proto file where you can outline particular props expected in a message, assign them to variables for the response, and then call them in your server and client files via the generated proto files based on your language.

Q3. **Unpacking the information received from a JSON encoded message**

```
import requests
from datetime import datetime

def getForecast(url):
   response = requests.get(url)

   if response.ok:
       forecast_json = response.json()
       today = forecast_json['properties']['periods'][0]
       todayDate = datetime.fromisoformat(today['startTime'])
       todayDateFormatted = todayDate.strftime('%m/%d/%Y')
       tomorrow = forecast_json['properties']['periods'][1]
       tomorrowDate = datetime.fromisoformat(tomorrow['startTime'])
       tomorrowDateFormatted = tomorrowDate.strftime('%m/%d/%Y')
       forecast_string = f'{today['name']} ({todayDateFormatted}) the expected
temperature is {today['temperature']}{today['temperatureUnit']} and {tomorrow['name']}
({tomorrowDateFormatted}) the expected temperature is
{tomorrow['temperature']}{tomorrow['temperatureUnit']}.'

       return forecast_string
   else:
       return 'There was an error fetching the data . . .'

def main():
   url = input('Provide a valid api.weather.gov gridpoint url (Hit enter to just do
NYC): ') or 'https://api.weather.gov/gridpoints/OKX/33,35/forecast'
   print(getForecast(url))

main()
```

Q4. **Reproduce the design using the HTML/CSS Box model**
I wasn't sure if the border was part of it, so I did end up adding an outerborder div.

```
<!DOCTYPE html>
<html>
   <head>
       <style>
           .outerborder {
               padding: 0.5rem;
               border: 1px solid black;
               width: 50%;
               position: relative;
           }
           .red {
               background-color: red;
               width: 100%;
               height: 200px;
           }
           .blue, .yellow {
               width: 50%;
               height: 50%;
           }
           .blue {
               background-color: blue;
           }
           .yellow {
               background-color: yellow;
           }
       </style>
   </head>
   <body>
       <div class="outerborder">
           <div class="red">
               <div class="blue">
                   <div class="yellow"></div>
               </div>
           </div>
       </div>
   </body>
</html>
```

Q5. Create a webpage with an HTML button

Again, was not sure if the border was part of it, so I added it just in case

```
<!DOCTYPE html>
<html>
   <head>
       <style>
           body {
               border: 4px solid black;
               min-height: 200px;
               padding: 0 2rem;
               border-radius: 2px;
           }
           #message_container {
               margin-top: 1rem;
           }
       </style>
       <script>
           function setBackground() {
               const body = document.body;

               if (body) {
                   body.style.backgroundColor = "orange";
               }
           }

           function setMessage() {
               const msg_container = document.getElementById("message_container");

               if (msg_container) {
                   msg_container.innerText = "Hello, World!"
               }
           }
       </script>
   </head>
   <body>
       <h1>Welcome to my webpage!</h1>
       <button onclick="setBackground();">Change Background</button>
       <button onclick="setMessage();">Display Message</button>
       <div id="message_container"></div>
   </body>
</html>
```

Q6. **The use of viewport in the context of mobile web page display**

There are a few techniques that can be used in order to adapt the display for mobile devices. One of those is to explicitly set the viewport. By setting the viewport in the fashion below, you are telling the browser to scale things based on the device width and considering the initial scale as 1 for when users have resizing/zooming enabled for accessibility.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Another method for supporting mobile is to use relative units instead of absolute units. Instead of measuring things by pixels, use measurements in em/rem, percentages, and vw/vh. It also is a good idea to consider using dvw and dwh as they tend to take into account the mobile browser header. These relative units allow for user resizing of text (em/rem) and also respecting device width much more consistently.

```
width: 100%;
font-size: 0.75rem;
height: 100dvh;
```

In terms of better formatting, flexbox and grid layouts are often more easily adapted for mobile web as they can operate off of the viewport dimensions. You need to make sure that you are utilizing flexbox appropriately (typically it isn't a good idea to set items to flex: 123px as that defeats some of the purpose of flexbox). In terms of grid layout, there is a frame measurement (fr) that breaks the viewport into n equal frames.

```
display: flex;
/* item inside */
flex: 1;
```

If you are going to be picky about changing layouts more dramatically, you can use CSS media queries, but that can get challenging to manage quickly especially if you separate out your concerns for components across multiple files. You need to be cautious about keeping these breakpoints properly in sync.

```
@media screen and (max-width: 768px) { … }
```

Q7. **Languages and Mobile Development Environments**

A. The standard language used to develop mobile Apps using Android Studio is Kotlin.
B. The standard language used to develop mobile Apps using Flutter is Dart.
C. The standard language used to develop mobile Apps using Xcode is Swift.

Q8. **AI-based code generators**

The first main concern that developers need to have when using AI-based code generators is the potentially faulty nature of the code. Generators tend not to have great context and can only attempt to generate code based on the information that you give it. In addition to that, the code it may generate can often have bugs, call upon libraries that you are not using, or be outright incorrect in solving your particular problem. It is best to thoroughly test all code that comes from code generators to ensure that they are both accomplishing the task it is intended to and that they account for all scenarios in which this code will be used.

Another concern developers need to have when relying on AI-based code generators is over-reliance on the code generator. If you do not understand what the code is doing, then when something inevitably breaks, you will not know how to fix it easily. Dependence on code generation can lead to ignorance of the codebase and long, often frustrating periods of debugging issues that you could more than likely have solved if you had been active in the development.