Section 1: Application Programming Interface

Part a

i. In GET method, usually it was used to request some particular data from a specific URL/resource. As it was said that GET is idempotent because it keeps providing the same result if we call it. For example, assuming I have a url, https://staging-api.com with an endpoint called /menu which returns a list of food inside a menu and also requires a user id for some validation/authentication which is implemented in the server before allowing the requested data to be sent back. I can call fetch('https://staging-api.com/menu?userId=123456') which will then returns me a set of data.

ii. In POST method, I usually use it when I want to create something. POST is not idempotent and provide different results because it creates something new. For example, assuming I have a url, https://staging-api.com with an endpoint called /menu and also requires user id for some validation. I want to create a new menu and store in the db. I can called something like this fetch('https://staging-api.com/menu?userId', body) whereby body = {method: 'POST', body: JSON.stringify(data)}

iii. Not sure on update method but I do know PATCH method if this is what is referring to. Usually used it to update or modify something. PATCH is not idempotent as it provides different end results if let's say item does not exist in the db then there is nothing to update but return an error. For example, I can update a food price in a menu by calling fetch('https://staging-api.com/menu/food/price', body) whereby body={method: 'PATCH', body: JSON.stringify(data)}.

iv. For PUT method, usually used it to update data too but if data does not exist that it will create a new data. PUT is idempotent cause the end result will always be having the same data changed no matter how much times we call on the endpoint.

Part b

We can use a token and pass into the header whereby the token may contains some personal data for verification. For example, we can use it such as fetch('https://staging-api.com', {method: 'GET', Authorization: 'Bearer abcdefg}) whereby abcdefg is a token. I am pretty sure other languages can add a token into their header, just different ways of adding it.

Part c

Format can be a full json format where is just contains the collection of data in it. My personal preference would be {status: '', data: {}} as it is cleaner and format stays the same. I can have status to validate whether it succeed or fail. Data will just be the collection of data.

Section 2: Simple Checkout System

Assuming I have a database in this format:

```
const data = {
  ipd: {
    name: 'Super ipad',
    price: 549.99,
    quantity: 0,
  },
  mbp: {
    name: 'MacBook Pro',
    price: 1399.99,
    quantity: 0,
  },
  atv: {
    name: 'Apple TV',
    price: 109.50,
    quantity: 0,
  },
  vga: {
    name: 'VGA Adapter',
    price: 30.00,
    quantity: 0,
  }
}
```

I will add a pricing rules function:

```
pricingRules = () => {
  let totalPrice = 0
  let scannedItems = []
  Object.entries(data).map(([key, value]) => {
    if (key === 'atv') {
      // Check if it is an apple tv
      if (value.quantity >= 3) {
        // if apple tv quantity more than or equals to 3 then only add a price of the reduction of the actual quantity by 1
        totalPrice += value.price * (value.quantity - 1)
      } else {
        totalPrice += value.price * value.quantity
      }
    } else if (key === 'ipd') {
      // check if it is an ipad
      if (value.quantity > 4) {
        //if ipad quantity more than 4 then add 499.99 only
        totalPrice += 499.99 * value.quantity
      } else {
        totalPrice += value.price * value.quantity
      }
    } else if (key === 'mbp') {
      // check if it is a mac book pro
      totalPrice += value.price * value.quantity
    } else if (key === 'vga') {
      // chceck if it is a vga adapter
      if (data[key].quantity > data['mbp'].quantity) {
        // if the quantity of vga adapter is more than macbook pro quantity then add the price of remaining vga
        totalPrice += value.price * (data[key].quantity - data['mbp'].quantity)
      }
    }
    const items = new Array(value.quantity).fill(key) // create array filled with the item based on the quantity
    scannedItems = scannedItems.concat(items) // join array to show SKUs scanned
  })
  return {
    'SKUs Scanned': scannedItems,
    'Total Expected': `$${totalPrice}`
  }
}
```

A class known as Checkout will do the calling for the function usage such as scan and total:

```
class Checkout {
  constructor (pricingRules) {
    this.pricingRules = pricingRules
  }

  scan (product) {
    if (data[product]) {
      data[product].quantity += 1
    }
    return
  }
  total () {
    return this.pricingRules()
  }
}
```