Postman Problem Set 3 Solutions

Useful examples of test scripts can be found here:
https://learning.postman.com/docs/tests-and-scripts/write-scripts/test-examples/

1 Response Handling

Problem 1.1: Validate that a ‘data‘ array is sorted by ‘timestamp‘ in descending order.

```
pm.test("Data is sorted by timestamp (desc)", function
   () {
    var timestamps = pm.response.json().data.map(item =>
        new Date(item.timestamp).getTime());
    var sorted = [...timestamps].sort((a, b) => b - a);
    pm.expect(timestamps).to.eql(sorted);
});
```

The json file is supposed to have the following form:

```
    {
  "data": [
    { "id": 1, "timestamp": "2024-12-01T10:00:00Z" },
    { "id": 2, "timestamp": "2024-11-25T09:30:00Z" },
    { "id": 3, "timestamp": "2024-11-20T08:00:00Z" }
  ]
}
```

Problem 1.2: Assert that all ‘events‘ objects have an ISO 8601 formatted ‘eventDate‘.

```
pm.test("All event dates are valid ISO 8601", function
   () {
    var regex = /^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(.\
       d+)?Z$/;
    pm.response.json().events.forEach(e => {
        pm.expect(e.eventDate).to.match(regex);
    });
});
```

The json file is supposed to have the following form:

```
    {
  "events": [
    { "eventId": "a1", "eventDate": "2025-01-01T14:23:00
       Z" },
    { "eventId": "b2", "eventDate": "2025-01-02T10:15:30
       Z" }
  ]
}
```

Problem 1.3: Ensure response includes a non-empty array and every object has more than 5 keys.

```
pm.test("Objects are well-formed", function () {
    var json = pm.response.json();
    pm.expect(json.records.length).to.be.above(0);
    json.records.forEach(obj => {
        pm.expect(Object.keys(obj).length).to.be.above
           (5);
    });
});
```

The json file is supposed to have the following form:

```
    {
  "records": [
    {
      "id": 101,
      "name": "Alice",
```

```
      "role": "Engineer",
      "email": "alice@example.com",
      "created_at": "2024-11-10T12:00:00Z",
      "active": true
    },
    {
      "id": 102,
      "name": "Bob",
      "role": "Manager",
      "email": "bob@example.com",
      "created_at": "2024-11-15T09:30:00Z",
      "active": false
    }
  ]
}
```

2 Chained Data Integrity

Problem 2.1: Compare ‘userId‘ from a previous request with one nested inside the current response.

```
pm.test("userId matches expected", function () {
    var expectedId = pm.environment.get("userId");
    var actualId = pm.response.json().meta.owner.id;
    pm.expect(actualId).to.eql(expectedId);
});
```

The json file is supposed to take the following form:

```
{
  "meta": {
    "owner": {
      "id": "user123"
    }
  }
}
```

Problem 2.2: For each ‘item‘, ensure its ‘details‘ sub-object includes a ‘category‘ and ‘tags‘ array with length ¿ 0.

```
pm.test("Items have complete detail structure", function
    () {
    pm.response.json().items.forEach(item => {
        pm.expect(item.details).to.have.property("
           category");
        pm.expect(item.details.tags).to.be.an("array").
           that.is.not.empty;
    });
});
```

The json file is supposed to take the following form:

```
    {
  "items": [
    {
```

```
      "id": 1,
      "name": "Item A",
      "details": {
        "category": "Electronics",
        "tags": ["tag1", "tag2"]
      }
    },
    {
      "id": 2,
      "name": "Item B",
      "details": {
        "category": "Furniture",
        "tags": ["tag3", "tag4"]
      }
    }
  ]
}
```

Problem 2.3: Validate that the user's 'subscription.endDate' is in the future.

```
pm.test("Subscription is still active", function () {
    var endDate = new Date(pm.response.json().
       subscription.endDate);
    pm.expect(endDate.getTime()).to.be.above(Date.now())
       ;
});
```

The json file is supposed to take the following form:

```
    {
  "subscription": {
    "endDate": "2025-12-31T23:59:59Z"
  }
}
```

3 Validation Against Custom Rules

Problem 3.1: Ensure no item in a list has a negative value in any numeric field.

```
pm.test("All numeric fields are non-negative", function
   () {
    const json = pm.response.json();
    json.items.forEach(item => {
        Object.entries(item).forEach(([key, value]) => {
            if (typeof value === 'number') {
                pm.expect(value).to.be.at.least(0);
            }
        });
    });
});
```

The json file should have this form:

```
    {
  "items": [
    { "id": 1, "price": 25.99, "quantity": 10 },
    { "id": 2, "price": 14.49, "quantity": 5 },
    { "id": 3, "price": 7.75, "quantity": 3 }
  ]
}
```

Problem 3.2: Assert all fields in ‘profile‘ object are non-empty strings.

```
pm.test("Profile fields are all non-empty strings",
   function () {
    const profile = pm.response.json().profile;
    Object.values(profile).forEach(value => {
        pm.expect(value).to.be.a("string").that.is.not.
           empty;
    });
});
```

The json file should have this form:

```
    {
  "profile": {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "bio": "Software Developer"
  }
}
```

Problem 3.3: Confirm every user has a unique combination of ‘email‘ + ‘username‘.

```
pm.test("Unique email-username combinations", function
   () {
    let seen = new Set();
    pm.response.json().users.forEach(user => {
        let combo = user.email + "|" + user.username;
        pm.expect(seen.has(combo)).to.be.false;
        seen.add(combo);
    });
});
```

The json file should have this form:

```
    {
  "users": [
    { "email": "alice@example.com", "username": "
       alice123" },
    { "email": "bob@example.com", "username": "bob123"
       },
    { "email": "charlie@example.com", "username": "
       charlie123" }
  ]
}
```

4 Cross-Field Logic and Templating

Problem 4.1: If user is ‘verified‘, their ‘verificationDate‘ must exist and be in the past.

```
pm.test("Verified users have valid verificationDate",
   function () {
    pm.response.json().users.forEach(user => {
        if (user.verified) {
            pm.expect(user.verificationDate).to.exist;
            const date = new Date(user.verificationDate)
               ;
            pm.expect(date.getTime()).to.be.below(Date.
               now());
        }
    });
});
```

The json file should have this form:

```
    {
  "users": [
    {
      "id": 1,
      "verified": true,
      "verificationDate": "2025-03-15T10:00:00Z"
    },
    {
      "id": 2,
      "verified": false,
      "verificationDate": null
    }
  ]
}
```

Problem 4.2: Ensure the sum of all ‘transaction.amount‘ values matches the ‘totalAmount‘ field.

```
pm.test("Transaction totals are accurate", function () {
```

```
    const json = pm.response.json();
    const sum = json.transactions.reduce((acc, tx) =>
       acc + tx.amount, 0);
    pm.expect(sum).to.equal(json.totalAmount);
});
```

The json file should have this form:

```
    {
  "transactions": [
    { "id": "TX001", "amount": 50.75 },
    { "id": "TX002", "amount": 25.50 },
    { "id": "TX003", "amount": 12.75 }
  ],
  "totalAmount": 89.00
}
```

Problem 4.3: Assert that each ‘comment‘ object includes the exact template keys: ‘id‘, ‘text‘, ‘user‘, ‘timestamp‘.

```
pm.test("Comment object structure validated", function
   () {
    const expectedKeys = ["id", "text", "user", "
       timestamp"];
    pm.response.json().comments.forEach(comment => {
        pm.expect(comment).to.have.all.keys(...
           expectedKeys);
    });
});
```

The json file should have this form:

```
    {
  "comments": [
    {
      "id": "C001",
      "text": "This is a great product!",
      "user": "user123",
      "timestamp": "2025-04-01T12:00:00Z"
```

```
    },
    {
      "id": "C002",
      "text": "Not bad, could be better.",
      "user": "user456",
      "timestamp": "2025-04-01T12:05:00Z"
    }
  ]
}
```

5 API Schema Enforcement

Problem 5.1: Validate that each item in the ‘feed‘ matches a manual schema using basic JS logic (e.g. title:string, score:number).

```
pm.test("Feed item schema validation", function () {
    pm.response.json().feed.forEach(post => {
        pm.expect(post.title).to.be.a("string");
        pm.expect(post.score).to.be.a("number");
        pm.expect(post.tags).to.be.an("array");
    });
});
```

The json file should have this form:

```
    {
  "feed": [
    { "title": "Post 1", "score": 98, "tags": ["tag1", "
       tag2"] },
    { "title": "Post 2", "score": 85, "tags": ["tag3", "
       tag4"] }
  ]
}
```

Problem 5.2: Assert that no duplicate ‘id‘ values exist across nested arrays in ‘categories[].items[]‘.

```
pm.test("No duplicate item IDs across categories",
   function () {
    let ids = [];
    pm.response.json().categories.forEach(cat => {
        cat.items.forEach(item => ids.push(item.id));
    });
    let unique = new Set(ids);
    pm.expect(unique.size).to.equal(ids.length);
});
```

The json file should have this form:

```
    {
  "categories": [
    {
      "category": "Electronics",
      "items": [
        { "id": "item001", "name": "Laptop" },
        { "id": "item002", "name": "Phone" }
      ]
    },
    {
      "category": "Furniture",
      "items": [
        { "id": "item003", "name": "Chair" },
        { "id": "item004", "name": "Table" }
      ]
    }
  ]
}
```

Problem 5.3: Validate that the current time (in UTC) is within the window defined by ‘startTime‘ and ‘endTime‘ in the response.

```
pm.test("Current time is within active window", function
    () {
    const now = new Date().getTime();
    const data = pm.response.json();
    const start = new Date(data.startTime).getTime();
    const end = new Date(data.endTime).getTime();
    pm.expect(now).to.be.within(start, end);
});
```

The json file should have this form:

```
    {
  "startTime": "2025-04-10T00:00:00Z",
  "endTime": "2025-04-15T23:59:59Z"
}
```