Episode 3 : Creating a Promise, Chaining & Error Handling

```
const cart = ['shoes', 'pants', 'kurta'];

// Consumer part of promise
const promise = createOrder(cart); // orderId
// Our expectation is above function is going to return me a promise.

promise.then(function (orderId) {
  proceedToPayment(orderId);
});

// Above snippet we have observed in our previous lecture itself.
// Now we will see, how createOrder is implemented so that it is returning a
promise
// In short we will see, "How we can create Promise" and then return it.

// Producer part of Promise
function createOrder(cart) {
  // JS provides a Promise constructor through which we can create promise
  // It accepts a callback function with two parameter `resolve` & `reject`
  const promise = new Promise(function (resolve, reject) {
    // What is this `resolve` and `reject`?
    // These are function which are passed by javascript to us in order to handle
success and failure of function call.
    // Now we will write logic to `createOrder`
    /** Mock logic steps
     * 1. validateCart
     * 2. Insert in DB and get an orderId
     */
    // We are assuming in real world scenario, validateCart would be defined
    if (!validateCart(cart)) {
      // If cart not valid, reject the promise
      const err = new Error('Cart is not Valid');
      reject(err);
    }
    const orderId = '12345'; // We got this id by calling to db (Assumption)
    if (orderId) {
      // Success scenario
      resolve(orderId);
    }
  });
  return promise;
}
```

Over above, if your validateCart is returning true, so the above promise will be resolved (success),

```
const cart = ['shoes', 'pants', 'kurta'];

const promise = createOrder(cart); // orderId
// ❓ What will be printed in below line?
// It prints Promise {<pending>}, but why?
// Because above createOrder is going to take sometime to get resolved, so pending
state. But once the promise is resolved, `.then` would be executed for callback.
console.log(promise);

promise.then(function (orderId) {
  proceedToPayment(orderId);
});

function createOrder(cart) {
  const promise = new Promise(function (resolve, reject) {
    if (!validateCart(cart)) {
      const err = new Error('Cart is not Valid');
      reject(err);
    }
    const orderId = '12345';
    if (orderId) {
      resolve(orderId);
    }
  });
  return promise;
}
```

Now let's see if there was some error and we are rejecting the promise, how we could catch that?
-> Using `.catch`

```
const cart = ['shoes', 'pants', 'kurta'];

const promise = createOrder(cart); // orderId

// Here we are consuming Promise and will try to catch promise error
promise
  .then(function (orderId) {
    // ✅ success aka resolved promise handling
    proceedToPayment(orderId);
  })
  .catch(function (err) {
    // ⚠️ failure aka reject handling
    console.log(err);
  });
```

```
// Here we are creating Promise
function createOrder(cart) {
  const promise = new Promise(function (resolve, reject) {
    // Assume below `validateCart` return false then the promise will be rejected
    // And then our browser is going to throw the error.
    if (!validateCart(cart)) {
      const err = new Error('Cart is not Valid');
      reject(err);
    }
    const orderId = '12345';
    if (orderId) {
      resolve(orderId);
    }
  });
  return promise;
}
```

Now, Let's understand the concept of Promise Chaining

-> for this we will assume after `createOrder` we have to invoke `proceedToPayment`

-> In promise chaining, whatever is returned from first `.then` become data for next `.then` and so on...

-> At any point of promise chaining, if promise is rejected, the execution will fallback to `.catch` and others promise won't run.

```
const cart = ['shoes', 'pants', 'kurta'];

createOrder(cart)
  .then(function (orderId) {
    // ✅ success aka resolved promise handling
    proceedToPayment(orderId);
    return orderId;
  })
  .then(function (orderId) {
    // Promise chaining
    // 💡 we will make sure that `proceedToPayment` returns a promise too
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    // from above, `proceedToPayment` is returning a promise so we can consume
using `.then`
    console.log(paymentInfo);
  })
  .catch(function (err) {
    // ⚠️ failure aka reject handling
    console.log(err);
  });
```

```
// Here we are creating Promise
function createOrder(cart) {
  const promise = new Promise(function (resolve, reject) {
    // Assume below `validateCart` return false then the promise will be rejected
    // And then our browser is going to throw the error.
    if (!validateCart(cart)) {
      const err = new Error('Cart is not Valid');
      reject(err);
    }
    const orderId = '12345';
    if (orderId) {
      resolve(orderId);
    }
  });
  return promise;
}

function proceedToPayment(cart) {
  return new Promise(function (resolve, reject) {
    // For time being, we are simply `resolving` promise
    resolve('Payment Successful');
  });
}
```

Q: What if we want to continue execution even if any of my promise is failing, how to achieve this?

-> By placing the `.catch` block at some level after which we are not concerned with failure.

-> There could be multiple `.catch` too. Eg:

```
createOrder(cart)
  .then(function (orderId) {
    // ✅ success aka resolved promise handling
    proceedToPayment(orderId);
    return orderId;
  })
    .catch(function (err) {
    // ⚠️ Whatever fails below it, catch wont care
    // this block is responsible for code block above it.
    console.log(err);
  });
  .then(function (orderId) {
    // Promise chaining
    // 💡 we will make sure that `proceedToPayment` returns a promise too
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    // from above, `proceedToPayment` is returning a promise so we can consume
```

```
using `.then`
    console.log(paymentInfo);
  })
```

Watch Live On Youtube below:

Edit this page