Promise Chaining:

Create UI applications with Flexible, Maintainable Flows

Shailendra Rai, SAS R&D India Pvt. Ltd, Pune
July 4, 2019

UI5con BANGALORE

learn.explore.connect.

About me

Shailendra Rai

Senior Software Specialist

SAS R&D India Pvt. Ltd, Pune

Work as UI developer for analytics products of SAS.

14 years of experience in enterprise application development both in UI and Java services side.

A major part of my work is to ensure, developed applications are flexible, maintainable and built for for long term support.

Other than work I like traveling to mountains, beaches and at times to spiritual places.

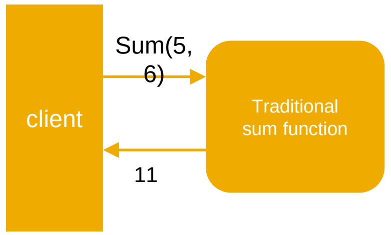
Recently I have been mostly occupied with my two kids and spending time with them.

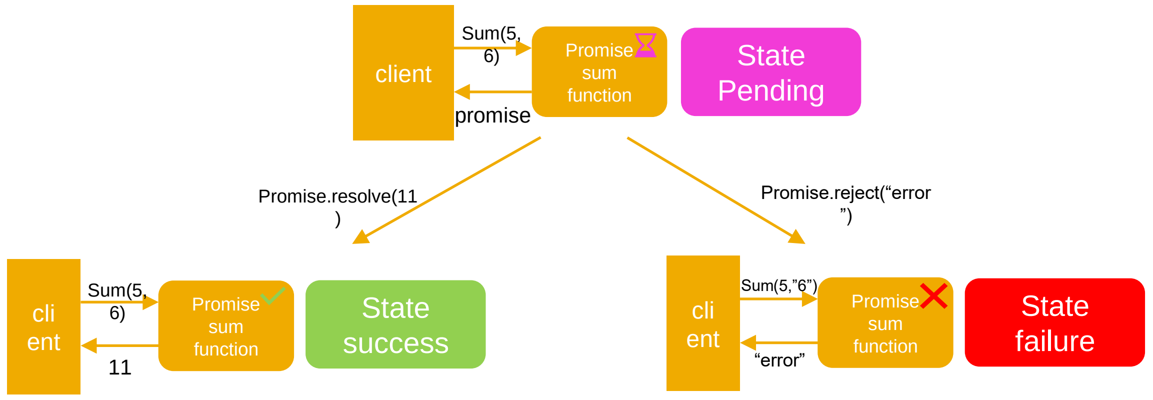
Agenda

- Promises as a concept.
- Identifying the need of a flow in the execution.
- Identifying the need of breaking down a complex task to atomic functions.
- Ways to have “flow-based execution” in traditional programming paradigms.
- Limitations of implementation with traditional approach.
- Ways to have “flow-based execution” with promises.
- Advantages of implementation with promises.

Promises with ES5 (using jQuery)

- ES5 doesn’t support promises as in-built feature. But ES6 supports promises
- jQuery provides objects for deferred and promises.
- Deferred states





Promises with ES5 (using jQuery)…

```
function sum(a, b){
    if(Number.isInteger(a) && Number.isInteger(b)) {
        return a+b;
    }else{
        return "one of the inputs: "+a+" OR "+b+" is not an
integer";
    }

}

console.log(sum(5,6));
console.log(sum(5,"6"));
```

```
// function definition
function sumWithPromise(a, b){
    var deferred = new $.Deferred(), that = this;
    // adding functional part in setTimeout, since we have to
    // return promise and real object needs to be resolved later
    var to = setTimeout(function(){
        if(Number.isInteger(a) && Number.isInteger(b)){
            deferred.resolve(a+b);
        }else{
            deferred.reject("one of the inputs: "+a+" OR "+b+" is not an
integer");
        }
        clearTimeout(to);
    });
    return deferred.promise();
}

sumWithPromise(5,6)
    .then(result => console.log(result),
        (error => console.error(error)));

sumWithPromise(5,"6")
    .then(result => console.log(result),
        (error => console.error(error)));
```

Advantages with Promises

Let's cook some food

- A complex set of execution of small processes.
- Easier of broken down in small functional units.
- Needs to be **Flexible**.
- Needs to be **Verbose**.
- Needs to be **Transactional**.
- Needs a **Flow** of execution
- Sample Application: https://github.com/shailenk/promiseAdvantages/

Credit: Photo by Maarten van den Heuvel from Pexels

Approach

Processer OpenUI5 ManagedObject:

- Traditional class: `demo.app.cooking.actions.CurryCookingSteps`
- Promise driven class: `demo.app.cooking.actions.CurryCookingActions`

Atomic functions sample:

```
washVeggies = function () {
    var to1 = setTimeout(function () {
        fireStepProcessed({
            // Fire even with event params
        });
        clearTimeout(to1);
    }, 200);
};
```

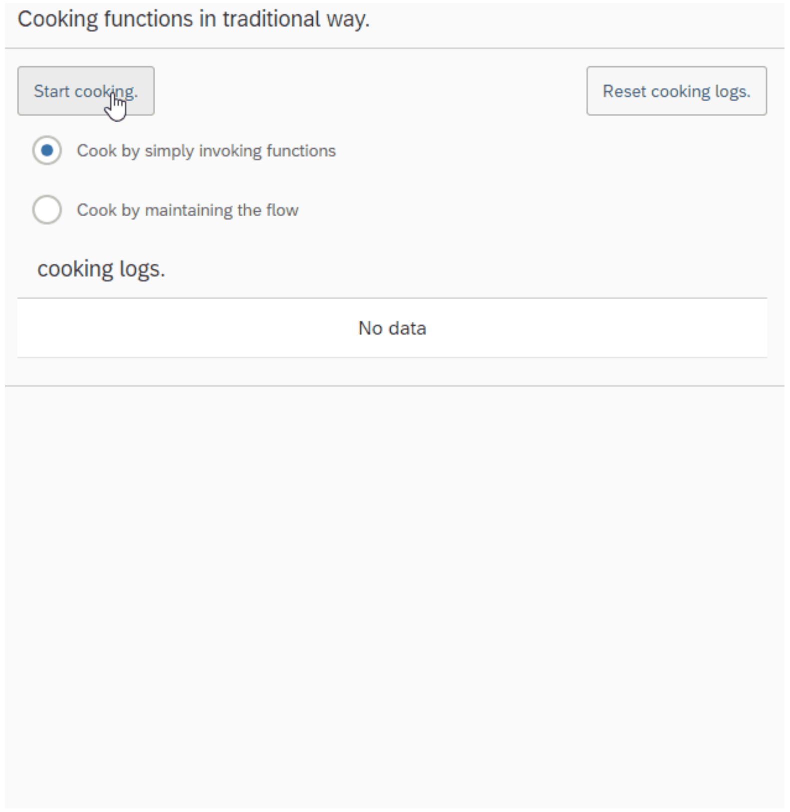
```
washVeggies = function () {
    var deferred = new $.Deferred();
    var to1 = setTimeout(function () {
        fireStepProcessed({
            // Fire even with event params
        });
        deferred.resolve();
        clearTimeout(to1);
    }, 200);
    return deferred.promise();
};
```

- Caller needs to listen to event: “stepProcessed”
- Atomic functions identified in this case:
  - **`washVeggies, cutVeggies, steamVeggies, precookSpices, mixAndCook, garnish`**

Traditional Approach 1

Call atomic functions one-by-one

```
cookTraditionally = function ()
{
    washVeggies();
    cutVeggies();
    steamVeggies();
    precookSpices();
    mixAndCook();
    garnish();
};
```


Cooking functions in traditional way.

[✓ Cook by simply invoking functions]
Cook by maintaining the flow

cooking logs.

No data

If atomic functions make use of async techniques like setTimeout, ajax calls. **No guarantee of flow-based execution**.

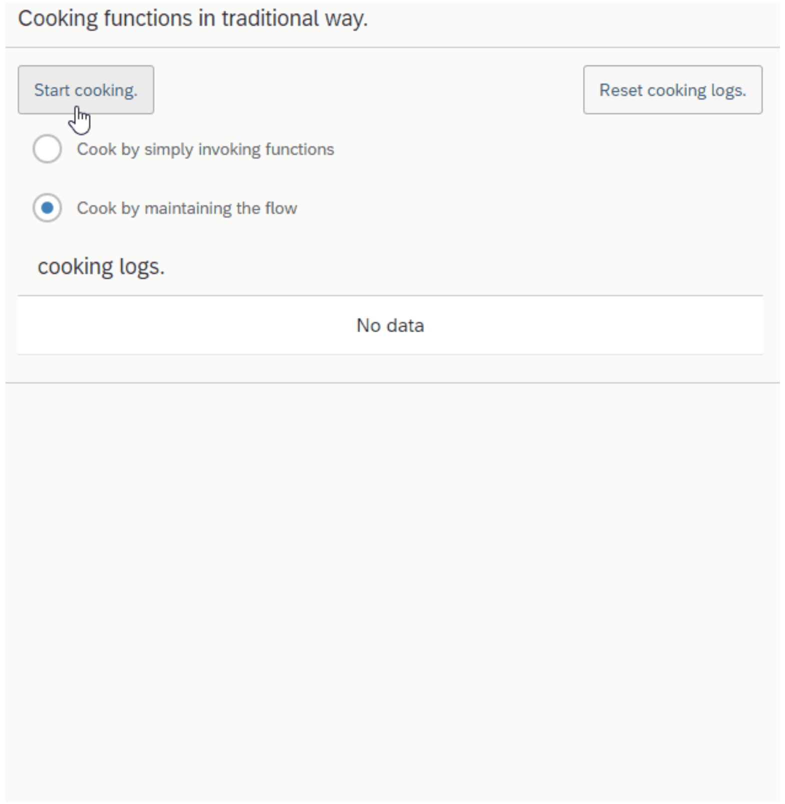
Promise → Flow → Atomicity → Traditional → Promise chain

Traditional Approach 2

Use events step execution

```
var seqenceArr =
[washVeggies,cutVeggies,steamVeggies,
    precookSpices,mixAndCook,garnish];

cookTraditionallyInSequence = function (seqenceArr) {
    var _currIndex = 0;
    attachStepProcessed(function () {
        if(_currIndex++ < seqenceArr.length - 1){
            seqenceArr[_currIndex].call(this);
        }
    });
    seqenceArr[_currIndex].call(this);
};
```


Cooking functions in traditional way.

( ) Cook by simply invoking functions
[✓ Cook by maintaining the flow]

cooking logs.

No data

- sequenceArr is an array of functions needed to be executed.
- How can we execute functionally independent steps like “steaming vegetables” and “pre-cook spices” in parallel?

Traditional Approach 3

Use callback functions

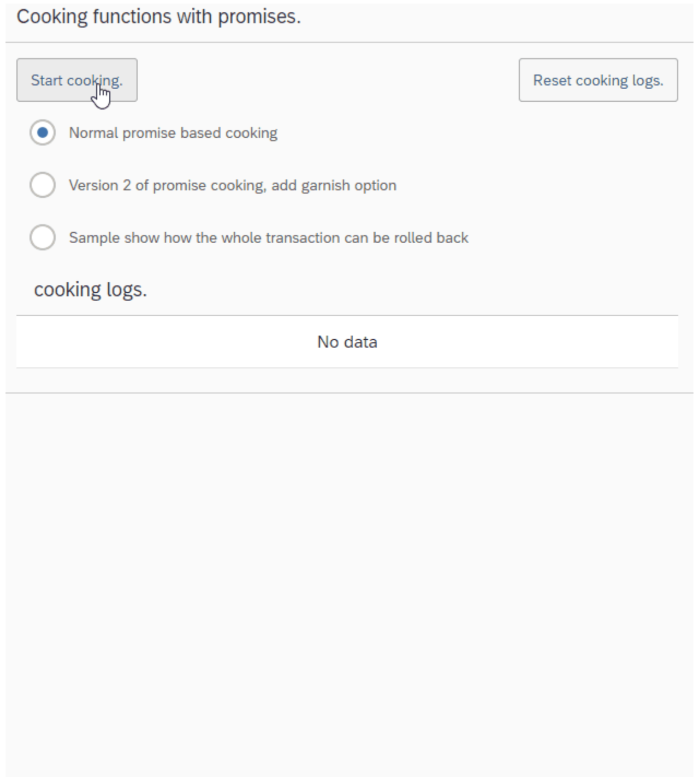
```
function mixAndCook(callBack5){
    fireStepProcess({
        //event params
    });
    callBack5();
}
function garnish(){
    fireStepProcess({
        //event params
    });
}
precookSpices(mixAndCook, garnish);
```

- Step execution and parallel execution can be achieved.
- Code is brittle
- Function definition needs signature of accepted callbacks, so **flow is rigid**
- Callback hell

Promise driven approach
Parallel execution and readability

```
cookWithPromises = function () {
    var deferred = new $.Deferred();
    function failHandler(errObj) {deferred.reject(errObj);}

    washVeggies().then(function (doneObj) {
        cutVeggies().then(function (doneObj) {
            $.when(steamVeggies(), precookSpices()).then(
                function () {
                    mixAndCook().then(function(){
                        garnish().then(function(){
                            console.log("last step done");
                            deferred.resolve("last step");
                        }, failHandler)
                    }, failHandler)
            }, failHandler)
        }, failHandler)
    }, failHandler);
    return deferred.promise();
};
```


Cooking functions with promises.

[✓ Normal promise based cooking]
Version 2 of promise cooking, add garnish option
Sample show how the whole transaction can be rolled back

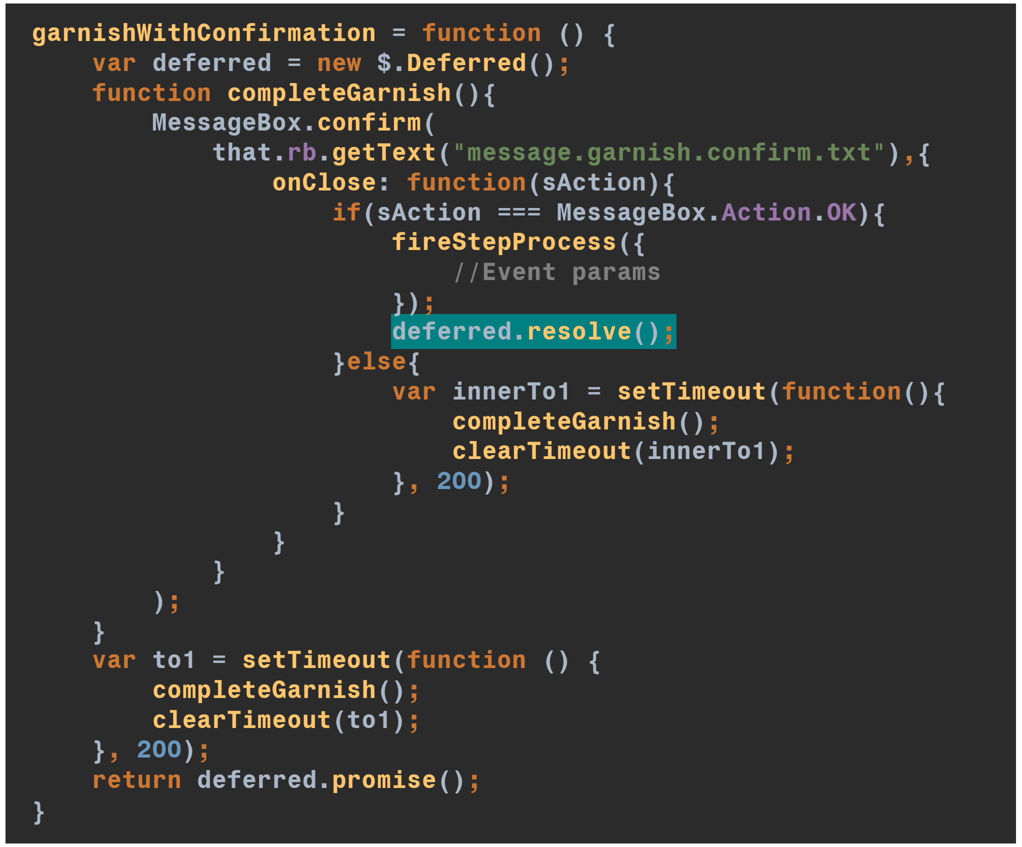
cooking logs.

No data

- Each success handler is calling next step. Flow is **Readable and Flexible**.
- With use if $.when, multiple promises can be executed in parallel.
- Observe in the logs, steps “steaming vegetables” and “pre-cook spices” are getting executed in parallel.
- Caller get better control.

Promise driven approach …

Resolve/Reject after actions on UI element. Eg: Adding a prompt to garnish function.



```
garnishWithConfirmation = function () {
    var deferred = new $.Deferred();
    function completeGarnish(){
        MessageBox.confirm(
            that.rb.getText("message.garnish.confirm.txt"),{
                onClose: function(sAction){
                    if(sAction === MessageBox.Action.OK){
                        fireStepProcess({
                            //Event params
                        });
                        deferred.resolve();
                    }else{
                        var innerTo1 = setTimeout(function(){
                            completeGarnish();
                            clearTimeout(innerTo1);
                        }, 200);
                    }
                }
            }
        );
    }
    var to1 = setTimeout(function () {
        completeGarnish();
        clearTimeout(to1);
    }, 200);
    return deferred.promise();
}
```

- Provide is free to decide when to resolve/reject.
- **Flexible** to add complex **reusable UI interactions**.
- **Maintainable** across versions.
- Wrapped actions ensure **Consistency**

Promise › Flow › Atomicity › Traditional › Promise chain

Promise driven approach …

Can easily add **Transactional** nature

```
garnishWithConfirmation().then(function(){
    MessageBox.confirm(
        that.rb.getText("message.garnish.confirmOrReject.txt"),{
            onClose: function(sAction){
                if(sAction === MessageBox.Action.OK){
                    deferred.resolve("last step done");
                }else{
                    deferred.reject("failed");
                }
            }
        }
    );
}, failHandler)
```

Handler in view file

```
cookWithPromisesWithRejectionPrompt().then(function(){},
function(error){
      sap.m.MessageBox.information(
         view.rb.getText("message.cookingNotAccepted.clear.txt"),{
            onClose: function(){
               //roll back the transaction
            }
         }
      );
   });
```

- Add **Transactional** nature to group of functions through promise driven "Wrapper/Higher-order" functions.

Demo Application

Summary

Shortcomings in traditional approach:

- Sequential execution is dependent upon implementation of atomic functions.
- Asynchronous execution of a group of atomic function is difficult.

Advantages with promises:

- Sequential execution is independent of implementation of atomic functions.
- Single/Grouped atomic functions can easily add interacting UI elements.
- Single/Grouped atomic functions can have transactional nature.
- Execution can be verbose.
- Sample shared at: promiseAdvantages sample

References

- Generic reading about promises and futures: https://en.wikipedia.org/wiki/Futures_and_promises
- Asynchronous programming in general: https://eloquentjavascript.net/11_async.html
- jQuery API page: https://api.jquery.com/deferred.promise/
- How browsers take advantage of jQuery promises: https://msdn.microsoft.com/en-us/magazine/gg723713.aspx
- Info on advancements in JS: https://www.youtube.com/watch?v=qbKWsbJ76-s
- All about ES6 approach on promises:
  - https://medium.com/@ramsunvtech/promises-of-promise-part-1-53f769245a53
  - https://medium.com/@ramsunvtech/js-promise-part-2-q-js-when-js-and-rsvp-js-af596232525c#.dzlqh6ski

Thank you.

Contact information:

Shailendra Rai
Senior Software Specialist
SAS R&D India Pvt. Ltd, Pune

Mail: shailendra.kumar@sas.com, maverick083@gmail.com
https://github.com/shailenk
https://www.linkedin.com/in/shailendra-rai-03789516/

BANGALORE
UI5con
learn.explore.connect.

Stretch Agenda

- Additional use case and sample showing reusability aspect.
- More about jQuery Deferred and Promises
- ES6 and other related topics.

Sample 2

Reusability aspect

- Appointment booking for simple tasks need dynamic prompts like time, address, equipment type etc.
- For a complex task like “Relocation of cable connection”, flow wise execution of sub-tasks might be needed.
- Example: For “Relocation of cable connection”, might need additional tasks of
    - Feasibility test at new address.
    - Logistics checks.
    - Electrical needs.
    - TV/Setup box repairs etc..
- The sample 2, shows how the functions written for simple tasks are used to provide flow for complex tasks.
- Sample shared at: https://github.com/shailenk/sample-work-order-docker

Difference between Deferred and Promise

- Deferred can be resolved, promise cannot.
- Promise doesn’t allow to change state.

```
function asyncEvent() {
    var dfd = jQuery.Deferred();

    // Resolve after a random interval
    setTimeout(function() {
        dfd.resolve( "hurray" );
    }, Math.floor( 400 + Math.random() * 2000 ) );

    // Reject after a random interval
    setTimeout(function() {
        dfd.reject( "sorry" );
    }, Math.floor( 400 + Math.random() * 2000 ) );

    // Show a "working..." message every half-second
    setTimeout(function working() {
        if ( dfd.state() === "pending" ) {
            dfd.notify( "working... " );
            setTimeout( working, 500 );
        }
    }, 1 );

    // Return the Promise so caller can't change the Deferred
    return dfd.promise();
}

// Attach a done, fail, and progress handler for the asyncEvent
var prom = asyncEvent();
prom.then((resp=> console.log(resp)),
    (error => console.error(error)));
prom.progress((notice=>console.info(notice)));
```

- Promise can be attached to a specific object

```
// Existing object
var obj = {
        hello: function( name ) {
            alert( "Hello " + name );
        }
    },
    // Create a Deferred
    defer = $.Deferred();

// Set object as a promise
defer.promise( obj );

// Resolve the deferred
defer.resolve( "John" );

// Use the object as a Promise
obj.done(function( name ) {
    obj.hello( name ); // Will alert "Hello
John"
}).hello( "Karl" ); // Will alert "Hello
Karl"
```

- Source: https://api.jquery.com/deferred.promise/

ES6 advancement

- ES6 has in-built promises.
- Makes use of event loop.
- Async await, keywords can make any normal function behave like promises.
- There is no deferred, but to achieve client-controlled action keywork yield is present.
- jQuery implementation and that in ES6 is very different.
- Promises as a concept is still evolving, but currently the one becoming popular is Promise/A+
- ES6 has promise A+ implemented which is still evolving.
- Implementors of A+ have listed big problems with PromiseA implementation which is also in jQuery: jQuery problems with promises
- Another good video from same guys: Info on advancements in JS, also talks about other es6 features.
- ES6 implementation is a mixture and a good page to refer: ES6 mix about promises
- Nice ref of when not to use jQuery
- With these basic to the advance features, its time to be watchful of the development.