# Defining Asynchronous JavaScript Code

JavaScript is, in most cases, single-threaded: in the browser, the thread executing JavaScript code will block the rendering process and any user interactions.

To avoid this blocking behavior, JavaScript calls that are expected to take some time, like doing server requests or capturing a photo through a mobile app plugin, have asynchronous APIs. This allows the code calling the API to get (and act upon) the result of the call later in time, when the result is available.

**Promises** allow you to execute, compose and manage asynchronous operations. They allow for asynchronous methods to return values like synchronous methods do. However, instead of the final value, the asynchronous method returns a promise for the value at some point in the future (or never).


## Define an Asynchronous Client Action

In an JavaScript element there are two predefined functions available for use: "$resolve()" and "$reject()". These functions are very similar to the usual "resolve()" and "reject()" functions used together with promises, but "$resolve()" doesn't take any arguments; output values should be set through the "$parameters" predefined object.

When one of "$resolve()" or "$reject()" predefined functions is used in a given JavaScript element, the client action the element belongs to will be considered asynchronous. In addition, if a given client action contains server calls, like executing server actions or refreshing aggregates, this will also make the client action to be considered asynchronous.

The generated code for an element that contains one of these two predefined functions is surrounded with the creation of a promise. Thus, there is no need to create a promise manually inside a JavaScript element code.
How to signal the end of an asynchronous client action to the caller

To signal a successful execution of the asynchronous client action being called, use the "$resolve()" predefined function.

For example, if you have a "GlobalAsyncAction" client action and you want to signal a successful execution to its caller, you can add the following code to a JavaScript element in the "success" path of the action flow:

```javascript
// Return success in GlobalAsyncAction client action
$parameters.Success = true;
$parameters.Out1 = 5.0;
$resolve();
```

The predefined function "$resolve()" doesn't take any arguments. To define output values in your client action, set them using the "$parameters" predefined object, as shown above.

You can use the same code construct to signal that an error occurred while performing the asynchronous client action. Following the example above, to signal an error to the caller in a "GlobalAsyncAction" client action, include the code presented below in a JavaScript element in the "failure" path of the action flow:

```javascript
// 1st approach: return failure in GlobalAsyncAction client action using an output parameter
$parameters.Success = false;
$parameters.ErrorMessage = "A custom error message";
$resolve();
```

Alternatively, you can call "$reject()" with an Error object, like in the following example. In this case, the error handling code defined through the promise's "catch(onRejected)" method (see the next section) will be executed, if defined:

```javascript
// 2nd approach: return failure in GlobalAsyncAction client action using $reject(...) $reject(new Error("A custom error message"));
```

If you do not include an Error object as an argument, a default Error object will be created, with a message describing the JavaScript element name where the "$reject()" method was called.

Only the predefined function "$reject()" will trigger an Exception in OutSystems, which will be handled by the first Exception handler available.

If you have not defined any custom exception handlers in your module, the exception will be handled by the default Global Exception Handler. Check Exception Handling Mechanism for more information.
Call an Asynchronous Client Action

When an asynchronous client action is called, its return value is not a simple JavaScript object, as in the synchronous case, but a promise. The output parameter values will be made available in the future (or eventually never) through the returned promise.

The code that acts upon these values must be included in a "then(onFullfilled)" method invocation, supported by the returned promise. The "onFullfilled" argument is a callback function that you should specify, like in the following example:

```javascript
// call the GlobalAsyncAction asynchronous client action
$actions.GlobalAsyncAction().then(function(result) {
    $parameters.Success = result.Success;
    if (result.Success) {
        $parameters.MyOutput = result.Out1;
        console.log($parameters.MyOutput);
    }
    // if the current client action is being invoked asynchronously, this call will resolve its promise
    $resolve();
});
```

The "result" parameter is an object that contains the output parameters of the called asynchronous client action, if any.

You can also act upon an unhandled error or a call to "$reject()" occurred in the asynchronous client action. To do that, do one of the following:

1. Add a second argument to the "then()" method. Its full signature is "then(onFullfilled, onRejected)", and the "onRejected" is a callback function called when "$reject()" is called in the asynchronous action, or
2. Chain the return value of the "then()" method, which is also a promise, with a "catch(onRejected)" method, that will handle the error case – this chaining process is called composition.

Example using "catch(onRejected)"

```javascript
// Using this approach, success/failure is signaled through the "Success" output parameter
$actions.GlobalAsyncAction().then(function(result) {
    $parameters.Success = result.Success;
    if (result.Success) {
        $parameters.MyOutput = result.Out1;
        console.log($parameters.MyOutput);
    }
    $resolve();
}).catch(function(err) {
    // err will contain an Error object, either due to an unhandled error or to $reject(...) having been called
    $parameters.Success = false;
    $parameters.ErrorMessage = err.message;
});
```
In the example above, the "catch()" callback function is encapsulating the error-related data as JavaScript element's output parameters, by setting "Success" to "false", saving the error message in the "ErrorMessage" output parameter and calling "$resolve()". In this case both "then()" and "catch()" callback functions end with a "$resolve()" call.

Another possible approach is to use "$resolve()" when the asynchronous call is successful, and use "$reject(...)

```javascript
// Example using the $resolve()/$reject(...) approach
$actions.GlobalAsyncAction().then(function(result) {
  $parameters.MyOutput = result.Out1;
  console.log($parameters.MyOutput);
  $resolve();
}).catch(function(err) {
  // err will contain an Error object, either due to an unhandled error in GlobalAsyncAction
  // or to $reject(...) having been called in the same action
  console.log("An error occurred in GlobalAsyncAction: "+ err.message);
  $reject(err);
});
```

Note that a "$reject(...)" call or an error not handled by a "catch()" will produce an error feedback message by default, shown at the top of your application screen:

![A custom error message](https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/JavaScript/Extend_Your_Mobile_and_Reacti...)

You should always include error handling code when calling asynchronous client actions in JavaScript using one of the two approaches presented above: "then(onFulfilled, onRejected)" or "catch(onRejected)". This error handling code will be used when the called asynchronous action invokes the "$reject()" method or when any unhandled JavaScript errors occur inside the called asynchronous client action.

Take into account that calling synchronous client actions in JavaScript must be coded differently than calling asynchronous client actions:

- When calling synchronous client actions the return value is a simple JavaScript object containing the output parameters.

- When calling asynchronous client actions a promise is returned, i.e. the returned values will not be available immediately after returning to the caller context. If you don't take this into account in your JavaScript code, the subsequent logic may be operating on invalid data.