Functional Component Class

Version:	1.0
Date:	March 20, 2020
Author:	Fred Dijkstra
Doc. id:	r7twd4e

Document history

Version	Date	Author(s)	Description
1.0	March 20, 2020	Fred Dijkstra (fred@oryxmovementsolutions.nl)	First complete version

Table of contents

Introduction

This document describes the Functional Component JavaScript class. This is a class that can be used to implement an event driven Node.js application, that is for example used in a wireless communication system.

A functional component - or *component* in short - implements a Finite State Machine that can be easily defined by the programmer to implement the required behavior for the application.

The class is implemented in the Node.js module `functionalComponent.js`.

State machine

A simple finite state machine is implemented as described in this chapter.

Definitions

A state machine[1] defines the behavior of the component by defining a structured deterministic behavioral model in which a component resides in a specific *state* and can only exit this state and move to another state due to an *event*.

Moving from one state to another state is called a *transition*. It is also possible to return to the original state. This is a self-transition.

A state machine can be defined by a state transition table comprising of the following columns:

- State
- Event
- Action
- Next state

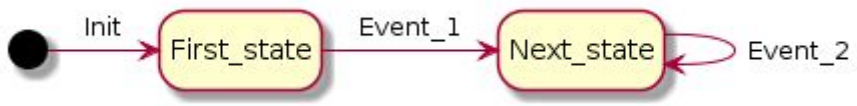
A graphical representation of such a state transition table is given by the state diagram as described in the following section.

State diagram

A *state transition table* is as given in the following:

State	Event	Action	Next state
Initial	Init	Component initialization.	First_state
First_state	Event_1	Some actions…	Second_state
Second_state	Event_2	Some actions…	Second_state

This is converted into the following state diagram[2].



The *first state* is defined by the initialization point, indicated by the black dot. This state can only be left after the component was initialized.

[1] http://en.wikipedia.org/wiki/Finite-state_machine
[2] This figure is drawn using PlantUML.

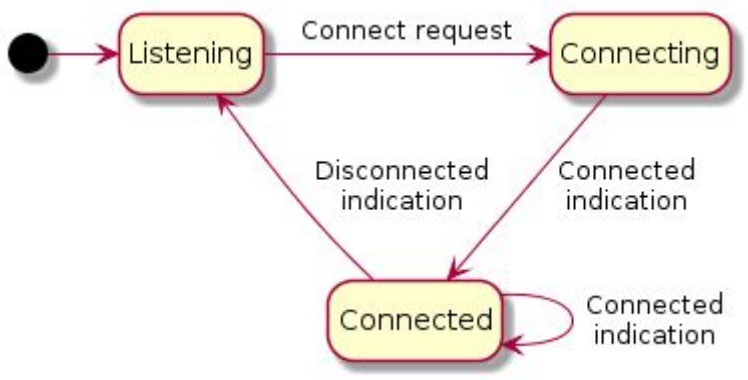
A transition between two states occurs when a certain event happens. It is also possible to have a self-transition in which the end-state of the event is the same as the begin-state.

As a simple example consider a master-slave wireless system in which the slave needs to connect to the master. To do this, the client regularly sends 'connection requests' and the master listens for these requests. When the master receives this request, it acknowledges it after which the slave responds with an indication that it is now connected. To monitor that the connection is still active, the slave re-sends this connected indication in regular intervals. When the connection is closed or lost, the master starts listening again.

For the master, this narrative specification can be translated into a *state transition table*.

State	Event	Action	Next state
Initial	Init	Component initialization. Start listening.	Listening
Listening	Connect request	Stop listening. Send connect acknowledgement.	Connecting
Connecting	Connected indication	-	Connected
Connected	Connected indication	-	Connected
Connected	Disconnected indication	Start listening.	Listening

This table can be depicted in the following state diagram.



Note that the state diagram does not reveal the actions that are to be performed on the transition.

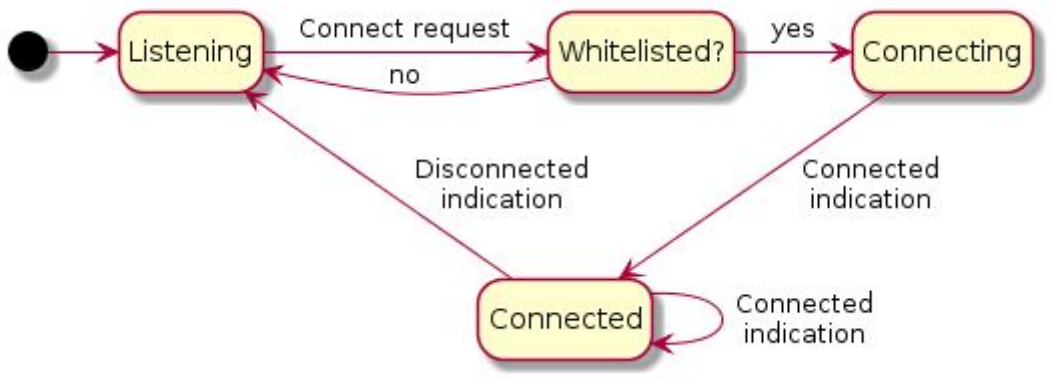
Choice-point

To indicate that a *decision* needs to be taken, the concept of a *choice-point* is used. A choice-point can be implemented by an intermediate state in which the state name is actually a question that can be answered with a simple 'yes' or 'no'. As such, in this state only two 'events' are handled; *yes* and *no*.

Although the choice-point is implemented as an intermediate state, it must not be possible that in this state other events can be handled than true or false and that no other messages are handled in this state. This will be ensured by the actual implementation of the state machine function (see "Execute transition").

As an example, consider an extension of the example in the previous section where the master only allows a client to connect when it is on a whitelist. This is implemented by entering a choice-point state in which it is checked.

As illustrated on the following (simplified) figure, the choice-point state is indicated by a question mark in the name and the possible 'answers' to this question.



Note that a choice-point is something that is used at specification time, but in essence is nothing more than making the decision visible in the state-diagram. Nothing will prevent the programmer to include an if-then-else statement in the transition code to obtain the same result. However, this is not preferable because it will obscure the behavior of the system.

Behavior specification

To be able to specify the behavior of a component in code, the state transition table and choice-points table can be specified as described in this chapter.

State transition table

The state transition table is an *array* containing objects with the following properties:

- state name
- event name
- next state
- transition function

Such a *state transition object* is defined as:

```
{
    stateName: '<state_name>',
    eventName: '<event_name>',
    nextState: '<state_name>',
    transFunc: function(component, parameters)
    {
         // transition function code
    }
}
```

In the table, a *choice-point* is indicated by the state-name ending with a question mark.

Choice-points table

When the state transition table contains choice-points, the corresponding evaluation function must be defined for each choice-point. This is done by supplying the *choice-points table*, which is an array containing objects with the following properties:

- choice-point name that ends with a question mark.
- transition function that returns a boolean (`true` or `false`).

Such a *choice-point object* is defined as:

```
{
    name:'<choicepoint_name>?',
    evalFunc: function( component )
    {
        return ( <evaluation> );
    }
}
```

Constructor

A Functional Component is constructed by supplying the *name*, the state transitions and the evaluation functions of the choice-points.

```
constructor( name, transitions, choicePoints )
{
    ....
}
```

The first step is to set the properties to their default values.

```
this.name               = name;
this.stateMachineTable  = {};
this.choicePoints       = {};
```

Next it must be checked whether the transitions table was defined and contains elements. If so, the supplied state transitions table is used as an argument for the `processStateTransitionsTable()` function. This function converts the table into a **`stateMachineTable`** object and returns this. As such the stateMachineTable property of the component can be set to the return value. By definition the state of the first transition is the first state.

```
if( this.transitions != undefined && this.transitions.length > 0)
{
    this.stateMachineTable = processStateTransitionsTable(transitions);
    this.currentState = this.transitions[0].state;
}
```

Also, in the same way the supplied choice-points table must be processed when it is defined and contains elements, using the `processChoicePointsTable()` function.

```
if (choicePoints != undefined && choicePoints.length > 0)
{
    this.choicePoints = processChoicePointsTable(choicePointsTable);
}
```

The `processStateTransisitionsTable()` and `processChoicePointsTable()` functions are discussed in the next chapter.

Initialization functions

Process state transitions table

As described in "State transition table", each entry of the state transition table is unique with respect to the state-event combination. To simplify the state machine implementation, the table is converted into an object **`stateMachineTable`** for which the named properties are the state-message combinations. This is done by calling the **`processStateTransitionTable()`** function which takes the state transition table as argument and returns a **`stateMachineTable`** object.

```
function processStateTransitionsTable( stateTransitionsTable )
{
    var stateMachineTable = {};
    ....
    return stateMachineTable;
}
```

The state transitions table is an array for which the elements can be evaluated[3]:

```
var entryString, transition;
for( var i=0; i < stateTransitionsTable.length; i++ )
{
    transition = stateTransitionsTable[i];
    ....
};
```

The **`stateMachineTable`** now gets a new named property for each state-message combination. While doing so, it is checked whether the combination was already present. If this is the case, an error message is displayed and the processing continues to the next entry.

```
entryString = transition.stateName + '-' + transition.eventName;
if( stateMachineTable[ entryString ] != undefined )
{
    console.log( "ERROR: transition for event '" + transition.eventName  + "' " +
                 "in state '" + transition.stateName + "' " +
                 "already defined!" );
    continue;
}
```

When successful, the transition can be added by setting the transition function that needs to be executed.

```
stateMachineTable[ entryString ] = transition;
```

[3] Use a for-loop to be able to go to the next element using the **`continue`** keyword.

Process choice-points table

When choice-points are defined for the component, then the choice-points table must be processed by calling the **`processChoicePointsTable()`** function which takes the component and choice-points table as arguments.

```
function processChoicePointsTable( choicePointsTable )
{
    var choicePoints = {};
    ....
    return choicePoints;
}
```

The only thing that is required is looping through the table and associating the evaluation function with the choice-points.

```
choicePointsTable.forEach( function(choicePoint)
{
    choicePoints[choicePoint.name] = choicePoint.evalFunc;
});
```

Execute transition

Whether the asynchronous action has a callback function or whether it actually generates an event, all events are handled by the **`eventHandler()`** method of the component.

This method has the event name and the parameters as arguments.

```
eventHandler( eventName, parameters )
{
    ....
}
```

The first step is to determine whether there is a transition for the state-event combination. If this is not the case, the event is unexpected and an error message is to be displayed while the event is ignored.

```
var entryString = this.currentState + '-' + eventName;
var transition  = this.stateMachineTable[entryString];
if( transition == undefined )
{
    console.log
    (
        "ERROR: component '" + this.name + "' " +
        "received unexpected event '" + eventName + "' " +
        "in state '" + this.currentState + "'"
    );
    return;
}
```

When the transition is present, it must be checked whether the *transition function* is defined and if not, the event is ignored after printing an error message.

```
if( transition.transFunc == undefined)
{
    console.log
    (
        "ERROR: component '" + this.name + "' " +
        "has undefined transition function for event '" + eventName + "' " +
        "in state '" + this.currentState + "'"
    );
    return;
}
```

When the transition function is defined, it can actually be executed, supplying parameters as the argument.

```
transition.transFunc( this, parameters );
```

After this is performed, the state can be set to the next state. The previous state is preserved so the state can be set back when anything fails in the next statements.

```
var previousState = this.currentState;
this.currentState = transition.nextState;
```

With the state set to the next state, it must be checked whether this state is a choice-point. By definition this is the case when the last character of the name is a question-mark. When the state is not a choice-point, handling is ready.

```
if( this.currentState.charAt(this.currentState.length-1) != '?' ) return;
```

When the next state is a choice-point, its *evaluation function* must be performed.

This means that first it must be checked whether the choice-point exists in the **`choicePoints`** variable. When this is not the case, the error must be handled meaning that an error message must be printed and the state of the component must be set to the previous state again.

```
var choicePoint = this.choicePoints[ this.currentState ];
if( choicePoint == undefined )
{
    console.log
    (
        "ERROR: component '" + this.name + "'  " +
        "unknown choice-point '" + this.currentState + "'"
    );
    this.currentState = previousState;
    return;
}
```

When the choice-point exists, then it can be executed which will return true or false. The **`eventHandler()`** method can be called recursively with the 'event' depending on the result of the evaluation function.

```
choicePoint(this)?
    this.eventHandler('yes'):
    this.eventHandler('no' );
```

Note that no data is passed, i.e. data will be undefined.

Also note that calling the function recursively means that the choice-point evaluation function is executed in the same handling of the initiating event.