venkateshcm.com

Reactor Pattern Part 3 - Promises to solve callback hell

Venkatesh CM

8-10 minutes

In Reactor Pattern Part 1 : Applications with Blocking I/O, I went through issues faced by a single threaded application to scale to handle more requests pre box and the corresponding issues it introduces. In Reactor Pattern Part 2 : Applications with Non-Blocking I/O I went through what Reactor Pattern is and how it fixes the Blocking IO issues and mentioned call back issue that async code introduces.

Lets get little more detail on what the callback issues are and introduce few options or libraries used to partially solve the issue, in this blog.

Example scenario

Consider a simple scenario where Restful service has to return a list of events for a given user. User is identified by Facebook Id. The service datastore has two tables or collections, (a) User table and (b) Event table which stores.

- user information
- id

- fb_id
- name
- ….
- events information.
- id
- user_id
- event_name
- event_desc
- start_date
- ….

  Assuming, you are not allowed to join user and event table or datastore is mongodb. The service controller will use below Pseudocode steps to get events.

- Given facebook id, get user id from User collection
- Given user id, get events for the user from events collection.

  Sequential Code with Blocking I/O

  In normal scenario the above steps directly translate to below code. If performance or scalability is not something playing on developers mind, Sequential code is the simplest and normal thing to do.

  Sequential Code With Blocking I/O

```
function getUserEvents(request,response){
  var facebook_id =
request.param('facebook_id');
```

```
  try{
      var user =
db.users.findOne({fb_id:facebook_id});
      var events =
db.events.find({user_id:user.id});
      response.write(events);
  }catch(err){
      response.status(500).send(err);
  }
}
```

There are religious wars between Ruby on Rails and NodeJS developers on Sequential Vs Evented style code.

- Key Points
- Majority of web applications (Ruby on Rails, Java Spring, Django etc) are written in sequential style.
- Sequential style is simple and readable.
- Most people think in sequential style i.e. Developers tend to break application logic into sequential steps like Pseudocode provided above.
- Boundaries of Pseudocode Step does not usually end at network call or IO call.
- Non-blocking I/O is considered when we need better scalabilty or performance.

Unfortunately Sequential code is linked to blocking I/O calls, becuase threads follow pre-emptive multitasking and not co-operative multitaking. More on this later when talking about Fibers.

Callback based solution

To solve Blocking I/O problem, code is split to three parts

1. Processing done before making a network or IO call
2. Network or IO call
3. Processing done after getting back data from network or IO call.

Execution flow for each of above steps is seperated and such that each of them can be executed from the event loop.

Non Blocking I/O with Call backs

```
function getUserEvents(request,response){

  var returnEvents = function(err,events){
          if (err)
respone.status(500).send(err);;
          response.write(events);
      });

  var givenUserFindAndReturnEvents =
function(err,user){
      if (err) respone.status(500).send(err);;

db.events.find({user_id:user.id},returnEvents);
  };

  var findUserAndReturnEvents = function(){
      var facebook_id =
request.param('facebook_id');
      db.users.findOne({fb_id:facebook_id},
givenUserFindAndReturnEvents);
  }

  findUserAndReturnEvents();
}
```

Notice that request and response objects are not passed to sub-functions. The sub-functions get access request and response since sub-functions are javascript closures. In fact if we move the sub-functions out-side getUserEvents method, it would not work. Which lead to chooice of keeping *givenUserFindAndReturnEvents* and *returnEvents* as sub-functions. Curring can be used to fix this problem, more on that in another blog.

Each of the sub-functions (*findUserAndReturnEvents*, *givenUserFindAndReturnEvents*, *returnEvents*) are executed asynchronously. functions *givenUserFindAndReturnEvents* and *returnEvents* are called call-back functions since they are triggered after getting back user object and event objects respectively from datastore.

The sub-functions could have been left as in-line or nested lamda functions. Nesting several such functions is another issue with call-backs.

- Key Points
- Code is separated based on pre-network call and post network call.
- The caller of the sub-function has to pass a callback function to execute after finishing sub-function task.
- Sequential logic is expressed asynchronously.

- Asynchronous code above is more scalable but may not be more performant (response time).
- Call-back causes readability issues – callback hell.
- Following execution flow is difficult, so called spaghetti-code.
- Non-Blocking API's impose major constrain on how you structure your code.
- Functions are hierarchy, i.e. calling function is responsible for functionality it provides as-well as the sub-function it calls. For example:– givenUserFindAndReturnEvents includes functionality of finding and returning Events to http response.

Promise based solution

To solve Call-back issues like spaghetti-code, we could use code structuring library like q promise. Promise library provides some code style standards and structuring, making it more readable compared to call-back based code shown above.

Non Blocking I/O with Promises

```
var loadEventsForUser = function(err,user){
  return db.events.find({user_id:user.id});
};

var findUser = function(){
  var facebook_id =
request.param('facebook_id');
  return db.users.findOne({fb_id:facebook_id});
}

function getUserEvents(request,response){
  var success = function(events){
          response.write(events);
  };

  var error = function(err){
      response.status(500).send(err);
  };

  findUser()
  .then(loadEventsForUser)
  .then(success)
  .fail(error);
}
```

Notice how code is split into smaller independent functions and how they are chained together using *.then* and *.fail* functions. Another important feature of promise library is how exception flow is handled. Compare promise based code with the call back based one above. Observe that errors are handled in each of the call back functions and when using promise, errors are isolated and handle seperately.

- Key Points
- Functions are flat, i.e. calling function is responsible for only its own functionality and can be used independently. For example :– findUser can be used independent of loadEventsForUser.
- Spliting sequential code into indenpendent functions which are reusable in multiple scenarios is not always easy. Many times functions are created just to work around Non-blocking reactive pattern.

- Functions can be used in other flows and could form reusable components.
- Better readability compared to call back option but not as simpile as sequential option
- Better exception handling compared to call back option but not as simple as sequential option.
- When libraries don't support promises, we end up writing boiler plat code to create promise and to handle async flows.

Conclusion

Async or Non-blocking IO introduces new challenges on how applications should to be structured and how async call backs can be abstracted away using promises like library. We need Non-blocking IO application since, sequencial blocking IO applications are not scalable. So Non-Blocking IO or Asynchronous code is not a desired feature but a nessesary evil to achive scalability.

Finally, assumption that Non-blocking IO and Asynchronous code are clubed together and one comes with other. Is it possible to get the best of both the worlds, i.e. Sequential code and Non-Blocking IO scalability. In fact, I think there is an option based on Fibers which can provide best of both the worlds. I will cover Fibers and how they achive both Non-Blocking IO and Sequential codebase in the Reactor Pattern Part 4 Write Sequential Non Blocking IO Code With Fibers In NodeJS.