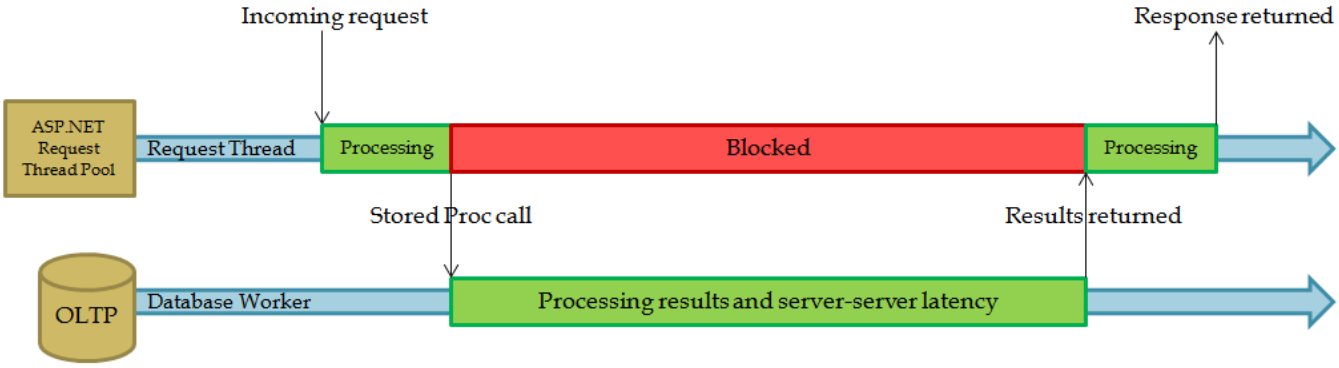
C# async/await usage

Why is synchronous code an issue with the V4 architecture?

In the Normandy architecture, we used the web application for processing requests, page events, formatting results, and returning html to the client. For anything involving the database, we'd synchronously call a stored procedure that would do some calculations and return several result sets to us:



There are several consequences of using this design:

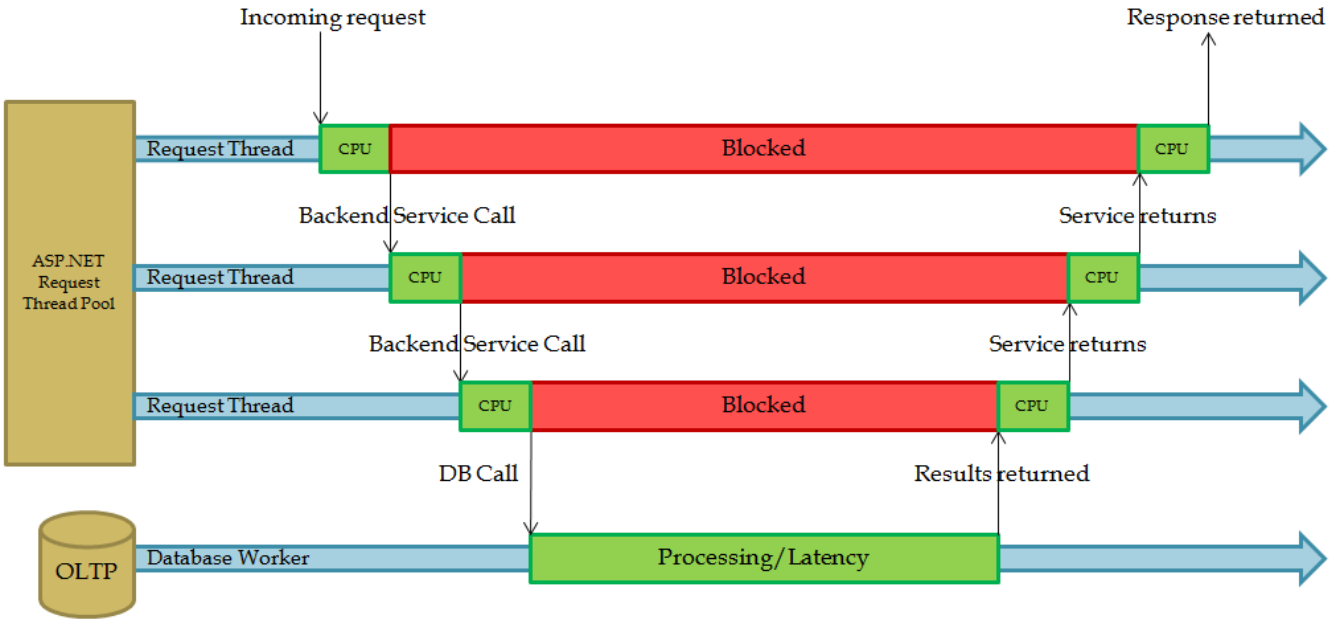
- It has the effect of tying up an ASP.NET worker thread for the duration of the request processing, even if that thread is blocked and doing nothing for 90% of the time. Threads have an overhead of ~1MB and are non-trivial to create/destroy, so the thread pool wastes memory keeping a lot of blocked threads in existence, and time creating a lot of threads to service blocking requests.
- If we get a big burst of web traffic, our servers won't respond well, since the ASP.NET Thread Pool only injects new threads at a rate of 2 a second (see bottom). This can be worked around by setting a higher value for minWorkerThreads, but that's not ideal.
- We rarely if ever made use of parallel database lookups, because they are hard to code and get right (either synchronously or asynchronously) in .NET 4.0 and earlier.

For the most part this wasn't a complete disaster, because memory is cheap to add, bursty load was fairly rare, and the need for parallel db lookups was also rare due to stored procedures returning multiple result sets.

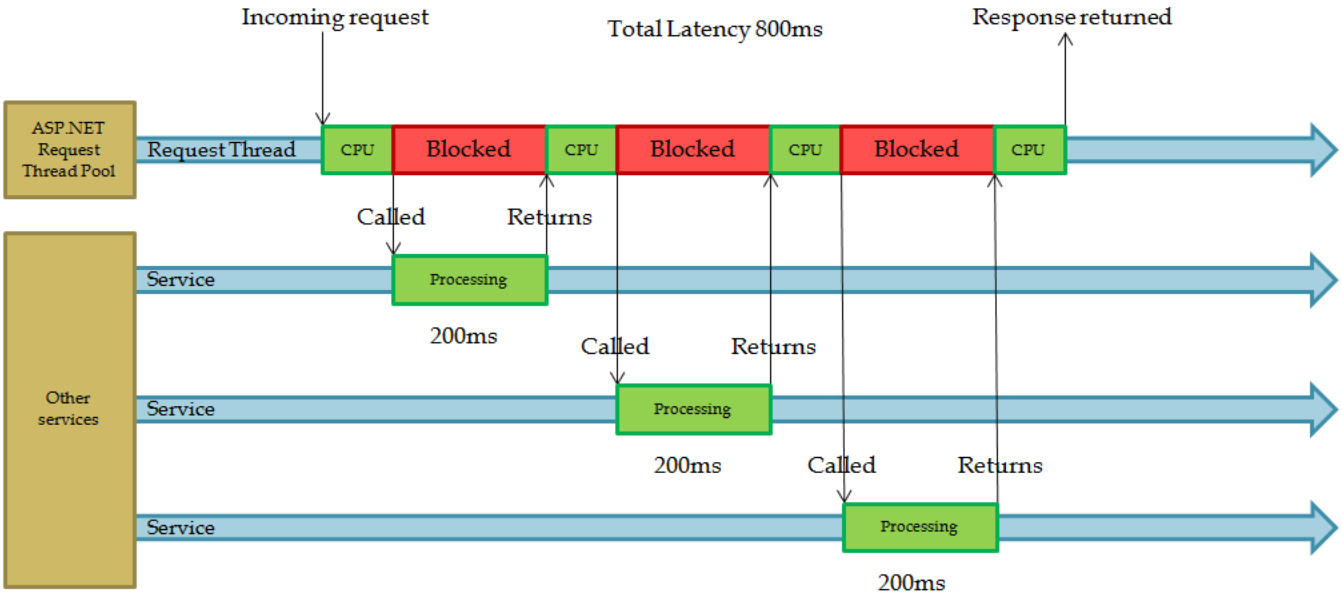
However, things change with our new V4 architecture:

- We no longer have a single monolithic database that we can query to get all of our results at once; the data is spread over multiple services.
- Because we're aggregating data from multiple services, there are a lot more cases where we could benefit from parallel service/data lookups.
- Since more services will be involved in fulfilling a single browser request, the number of threads that could end up blocked by synchronous I/O goes up, further causing thread pool issues.

In this example, we're dealing with the issue of lots of threads being spawned just to deal with blocking I/O calls:



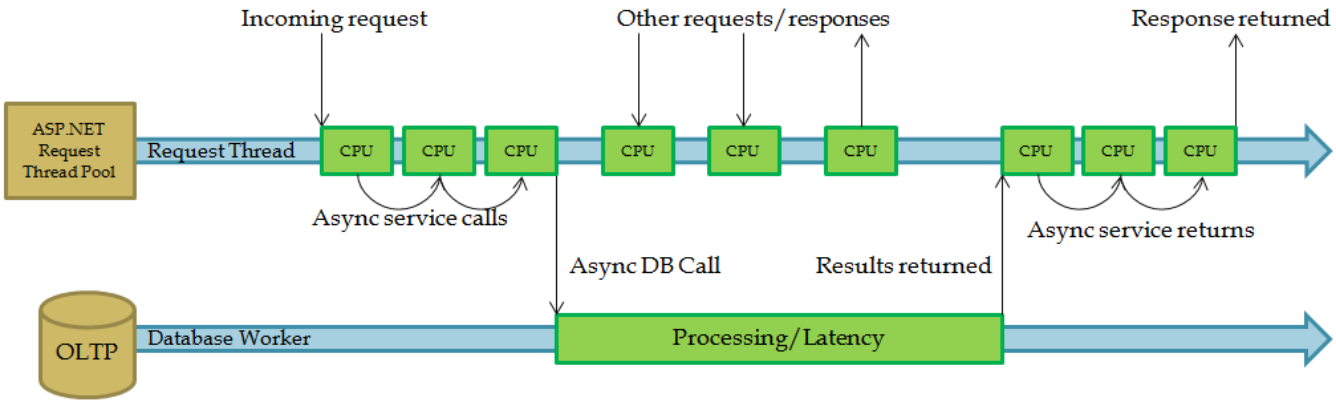
In this example, we're dealing with increased overall latency due to failure to parallelize service/data calls:



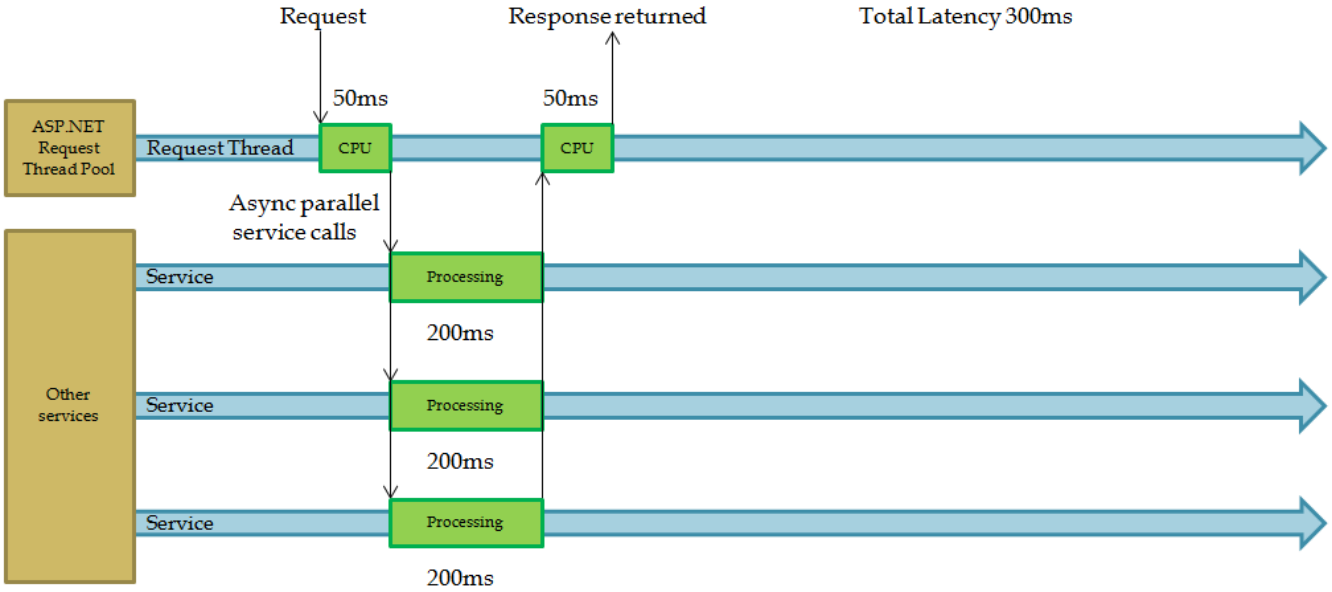
How does asynchrony change things?

If we make our calls asynchronous, it helps us by freeing up ASP.NET worker threads when they're not in use. This allows individual machines to get higher throughput when servicing requests, and we can also parallelize calls that are unrelated to each other.

In this example, I/O doesn't block, so a single worker thread can service multiple requests in parallel:



In this example, we can make unrelated service calls in parallel to reduce our overall response time:



Using async/await in .NET 4.5 to make our life easier

The new keywords available in C# 5 vastly simplify making asynchronous calls. Rather than setting up continuations or separate callback methods, you can write your code almost exactly the same as you did before, with the addition of two new keywords:

```
public async Task<ActionResult> GetAttendeesForEvent(Guid eventStub)
{
        var attendeeClient = new AttendeeClient();

        List<Attendee> attendees = await attendeeClient.GetForEventAsync(eventStub);

        var viewModel = BuildViewModel(attendees);
        return View(viewModel);
}

...
public interface IAttendeeClient
{
        Task<List<Attendee>> GetForEventAsync(Guid eventStub);
}
```

There are a few things to note about this code:

- The method is marked as 'async'
- Even though GetForEventAsync returns a type of Task<List<Attendee>>, the assignment to the attendees variable ends up as List<Attendee> because of the 'await' keyword.
- The method returns a type of ActionResult in the body, even though the return type is Task<ActionResult>

Make sure to read about Potential deadlocks with async/await before using async/await

So what exactly is going on?

(mention how the compiler turns the method into a state machine w/ automatic continuations)

(explain how await 'pauses' the method if the result is not available)

(explain how returning from the method wraps the value in a task)

It's also trivially easy to fire off multiple requests in parallel, and then just wait for them all to finish before continuing to execute code:

```
public async Task<ActionResult> GetOverallEventDetails(Guid eventId)
{
        Task<EventDetails> eventDetailsTask = _eventClient.GetEventDetailsAsync(eventId); //Takes 300ms
        Task<EventReg> eventRegTask = _eventRegClient.GetEventRegAsync(eventId); //Takes 300ms
        Task<EventBudget> eventBudgetTask = _eventBudgetClient.GetEventBudgetAsync(eventId); //Takes 300ms

        EventDetails eventDetails = await eventDetailsTask;
        EventReg eventReg = await eventRegTask;
        EventBudget eventBudget = await eventBudgetTask;

        //Only takes ~300ms
        var viewModel = BuildOverallEventDetails(eventDetails, eventReg, eventBudget);

        return View(viewModel);
}
```

(show async EF example)

(show thread-specific stuff (Request) being accessed after an await)

(show how exceptions are thrown)

Best Practice for Async/Await pattern in ASP.NET WebForms

Still there are portions of our Normandy legacy code base that are on ASP.NET WebForms. If you plan to use asynchronous pattern in WebForms or are forced to use the pattern because you are consuming a client library which requires it then you have to use it throughout the request life cycle. Not using this pattern properly in WebForms can cause deadlocks where end users will notice a white screen and page stuck in processing. Below is an example.

Always mark your page directive as Async.

SomePage.aspx

```
<%@ Page Async="true" Language="C#" AutoEventWireup="true" CodeBehind="myAsyncPage.aspx.cs" Inherits="Cvent.Web.
Subscribers.Account.myAsyncPage" %>
```

Note it is not recommended to use async/await solely in a web control (.ascx) page as this directive would be marked as invalid.

Register a new async task and always avoid declaring method as "private async void" even when IDE suggests you to do so.

myAsyncPage.aspx.cs

```
        protected void Page_Load(object sender, EventArgs e)
        {
                RegisterAsyncTask(new PageAsyncTask(DoMyAsyncTask));
        }

        private async Task DoMyAsyncTask()
        {
                                var externalCallResult= await MakeExternalCall();


                                if(externalCallResult.StatusCode == HttpStatusCode.OK)
                                {
                                        //do something
                                }
                }




        private async Task<ExternalCallResponse> MakeExternalCall()
        {
                        var externalCallResponse =  new ExternalCallResponse();
                        var api = new Api("key","url");

            try
            {
                externalCallResponse = await api.Method.Call();
            }
            catch (Exception ex)
            {
                throw new Exception(ex);
            }

            return ExternalCallResponse;


        }


#if you plan to redirect user in an async page you should allow page to continue execution by setting Response.
Redirect("/uri",false); where default it true which results in thread abort exception.
```

Mention guidelines

- use async throughout the entire stack
- avoid deadlocks related to synchronization context
- also include configureAwait usage

Mention additional resources

- Microsoft async/await best practices
- MSDN async/await introduction article
- Video on async/await in ASP.NET; also has a brief history on asynchrony in the .NET framework