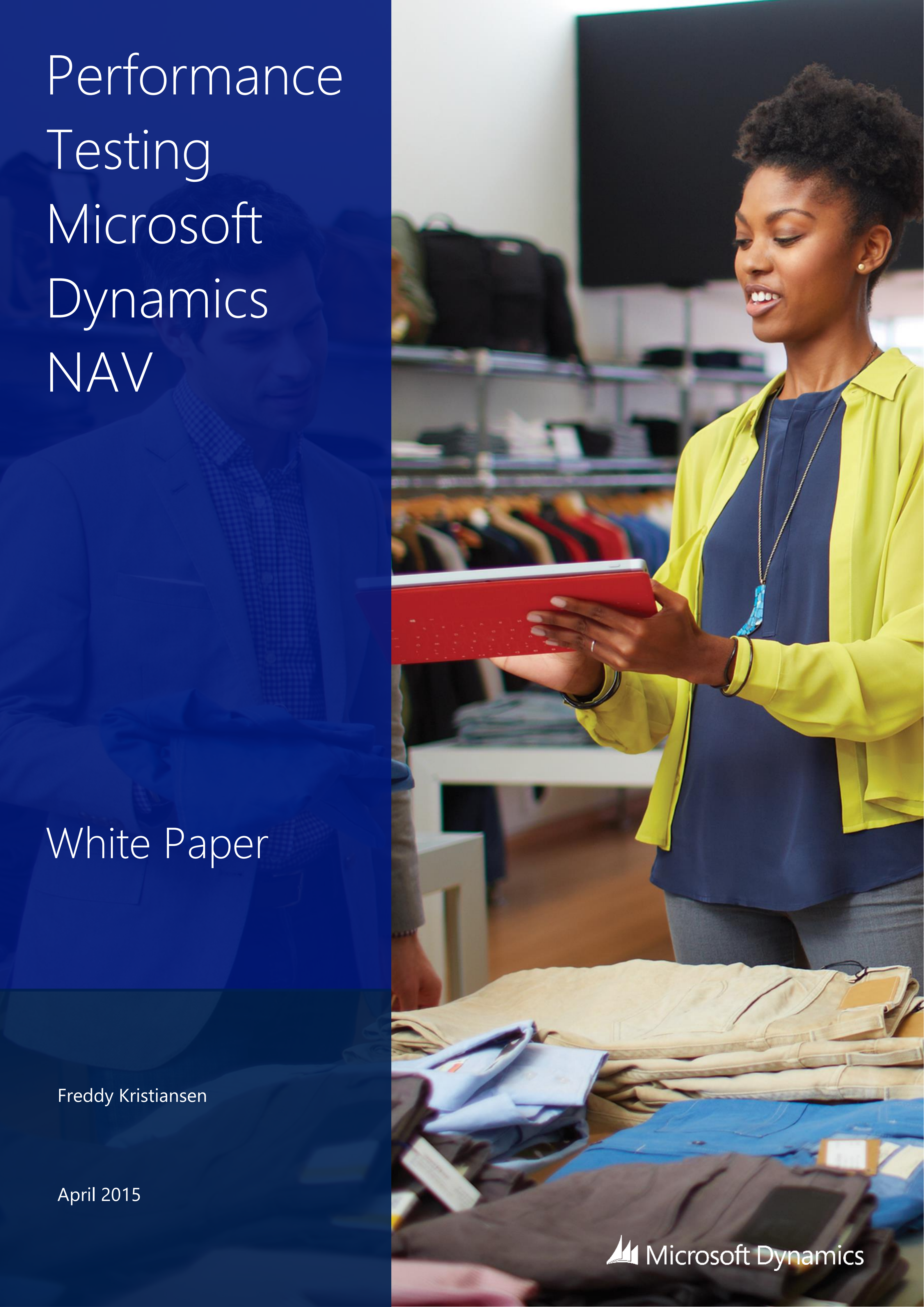
Performance Testing Microsoft Dynamics NAV

White Paper

Freddy Kristiansen

April 2015



Contents

Overview

This white paper provides detailed insight into the concepts and principles that support performance testing in Microsoft Dynamics NAV. By outlining some sample goals from the start, it explains how you can use load testing to optimize your application and hardware to meet the customer and user requirements – addressing typical questions around the number of users or tenants, hardware requirements, scaling, and upgrading.

The documentation starts by describing conceptual overviews of perquisites that one should have in order to work with performance testing in Microsoft Dynamics NAV.

Then it explains the technical architecture to show a "sample" goals, which might be similar to yours and which all can be achieved by running performance tests.

Finally, it provides detailed description on performance testing processes.

Introduction

One of the most frequent questions I hear when talking about Microsoft Dynamics NAV in a hosting environment is: "How many tenants or how many users can you fit on one NAV service tier?"

While I would love to be in a situation where I could reply 42 or any other fixed number, this would really be the same as answering the question of how many people can you fit in a bus or how many files can you put on a hard drive.

In reality – it depends - it depends on the scenarios the users are doing, the application they are using, and the size of the server (memory + CPU performance).

This probably doesn't come as a surprise if you think about it.

When a user connects to a service tier, the service tier will allocate some memory to hold the session state (open pages, global variables, permissions, etc.). This memory won't be released until the user disconnects or logs out.

If the user is performing a task like creating a sales order, then the service tier might have to allocate some more memory, and it will also have to spend some CPU cycles on opening pages, running business logic, reading data, updating data, and so on.

In our existing white papers about sizing guidelines for multitenant deployments and for on-premise deployments, we used a set of pre-defined scenarios geared towards the W1 application on a set of pre-defined hardware configurations. This may resemble your typical user and could maybe be used as an indicator of where to begin.

Note though, that one single line of non-optimal customized code in the application, a different user type, or another hardware profile, can change the picture dramatically and if you want to have more accurate measurements, you need to run custom performance tests yourself, simulating the scenarios on the application and on the hardware you want to use.

Assumptions

This whitepaper assumes that you are familiar with the following concepts:

- Deployment of Microsoft Dynamics NAV 2013 R2 and Microsoft Dynamics NAV 2015
- C#

You should also be familiar with the following whitepapers:

- Microsoft Dynamics NAV 2013 R2 on Windows Azure
- Microsoft Dynamics NAV 2013 R2 Sizing Guidelines for Multitenant Deployments
- Microsoft Dynamics NAV 2013 R2 Single Tenant On-premise Sizing Guidelines
- Microsoft Dynamics NAV 2013 R2 Large Scale Hosting on Windows Azure

Who is the audience of this whitepaper?

This whitepaper is for the partners or customers planning and/or implementing performance tests for Microsoft Dynamics NAV 2013 R2 or Microsoft Dynamics NAV 2015.

Before starting a project to run performance tests, you need to define the goal. Below are a number of "sample" goals, which might be similar to yours and which all can be achieved by running performance tests. You should define your goal and what you want to get out of your efforts. Maybe you have multiple goals, in which case, you might find yourself writing different performance tests for different goals, but using the same execution engine.

Can Microsoft Dynamics NAV handle "x number of" users on a single tenant system?

In this case, you probably already know the scenarios and the approximate frequency of the individual tasks. You are looking for two things: one is the hardware requirements, and two is whether you would need to optimize your application in order to avoid locking or other application-related issues with this number of users.

Running performance tests that simulate the right number of users doing the predefined scenarios can definitely help you achieve this goal.

What are the hardware requirements for running my vertical solution in a multi-tenant environment for "x number of" customers with "y number of" users per customer?

In this case, you probably have an idea of the most common scenarios and their approximate frequency. You are looking for insurance that Microsoft Dynamics NAV can handle this performance, and you want to know the cost of running a setup like this. You may also want to know whether there are any application-related issues which cause severe load on the either the service tier or the SQL Server.

Running performance tests that simulate the number of users spread over the right number of tenants will help you identify the hardware requirements.

How well does my vertical solution scale in a multi-tenant environment when adding customers and users?

In this case, you probably also have an idea of the most common scenarios and their approximate frequency. You are looking to learn when you need to scale up or out on the service tier and/or on the SQL Server. You may also want to know whether there are any application-related issues which cause severe load on the either the service tier or the SQL Server.

Running performance tests that simulate an increasing number of users can help you identify the user limit on a service tier. In the test environment, you can re-run your tests with additional service tiers to identify the limit the SQL Server. You can also run tests spreading tenants over two or more SQL Servers.

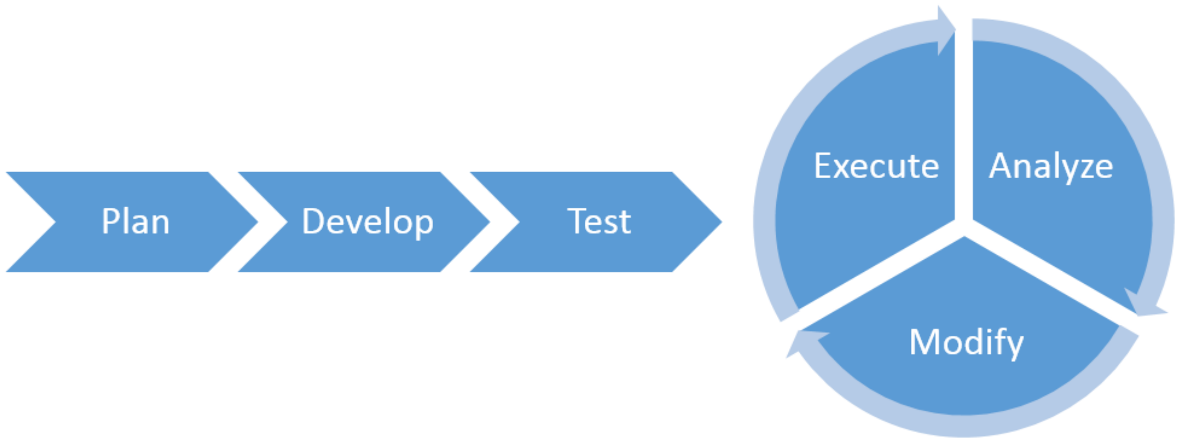
When upgrading my solution to the next cumulative update, does it still allow the same number of customers and/or users?

In this case, you have created and run performance tests against your solution for one of the above reasons, and you want to make sure that upgrading or changing code in your application doesn't change this picture significantly.

Keeping the performance test environment intact and re-running with every upgrade will help you achieve this.

Performance testing process

Developing performance tests might be a straightforward process. You plan, develop, and test the performance tests, and then you are done. When running the performance tests the first time, you probably have an expected outcome. You have a good idea of how well the solution scales and/or where the bottlenecks are. When analyzing the results you might find that your expectations weren't met. In which case, it is a good idea to investigate why, identify the bottlenecks, modify the app and/or the infrastructure, and run the tests again.



When you are satisfied with the results or you have accepted them as they are, you have established a baseline for your application and have a better understanding of the bottlenecks you should compare and analyze every subsequent execution of the performance tests with this baseline and understand why numbers change.

Plan

The value you get from running performance tests depends largely on the investment you make in identifying the types of usage and what scenarios they perform.

Identify scenarios and user types

Think about the different usages of your system and try to categorize them.

1. Users who are working in Microsoft Dynamics NAV all day, starting the system in the morning and logging out again in the afternoon after a good days work. This is kind of the traditional Microsoft Dynamics NAV user.
2. Warehouse workers doing pick, pack, and ship.
3. Point Of Sale system users.
4. Users who access data from Microsoft Dynamics NAV by using the SharePoint client through an intranet portal and occasionally access the Microsoft Dynamics NAV Web client to query more information. These users primarily read data and use a set of specific tasks available in the portal for entering data.
5. Users who access Microsoft Dynamics NAV from a tablet client for some BI and running some reports to see how the business is doing.
6. Applications (like web-shops) that access Microsoft Dynamics NAV through web services and add orders to your system automatically.
7. And more.

Having mapped out the different kind of usages, you need to go through the actual scenarios. For each of the scenarios, I suggest recording a video which shows a user performing the scenario at a real-life pace.

For application scenarios using web services, I suggest logging actual usage over time in order to know the level of load of these applications.

As the final preparation, create a map over the types of usage and the scenarios with an average frequency per hour, for example:

- User1
  - Create sales order (2 times per hour)
  - Create purchase order (2 times per day)
- User2
  - Use SharePoint Portal with Microsoft Dynamics NAV web parts (refresh 6 times per hour)
  - Invoke time registration task (2 times per day)
- User3
  - …

Develop

In order to run scenario tests on Microsoft Dynamics NAV, we need to emulate real Clients running. As you know, Microsoft Dynamics NAV has the ability to run all business logic from different clients like Windows, Web and Tablet. All of these clients are utilizing the same logical client backend.

Client Service

This logical client backend is exposed as a web service, called *client services*, and we will use this service to run our performance tests. This means that the load on the Microsoft Dynamics NAV service tier will be identical to a user performing these tasks from the Windows client. The main difference is that no actual rendering will happen. Rendering on the Windows client will happen on the client and won't affect the service tier. Instead of the rendering, we will insert delays, in order to make the scenarios run at an actual pace.

The client service is exposed on the Web client and thus you will have a close to similar load on the IIS Application using the client service when compared to using the Web client. The client service will not render HTML like the Web client, so we will assume that the CPU power needed for the HTML rendering is insignificant compared to the remaining things happening.

Note that when running Microsoft Dynamics NAV 2013 R2 with Forms Authentication, please consult the comments in the web.config file and make sure that you have uncommented the section that allows access to the ClientService.svc to users without redirecting them to the sign-in page (not needed for Microsoft Dynamics NAV 2015 or later).

GitHub

You will find a couple of projects on GitHub, which contains some sample code on how to do performance tests. The GitHub organization is used to store the sample repositories and is available here:

https://github.com/NAVPERF

On a high level, a solution using client services will establish a connection (which resembles starting the client), and then the session will be idle.

Every time the client wants to do something with the session, it needs to invoke a *ClientInteraction*. A ClientInteraction resembles an action that the user is performing. This can be clicking an action, activating a control, closing a page, and a lot of other things.

At the end of the scenario, you close the session (which resembles closing the client).

The GitHub project contains a couple of end to end samples.

Connecting to Microsoft Dynamics NAV

Establishing a connection to Microsoft Dynamics NAV and opening a session is done using these lines:

```
var context = new UserSession.UserContext();
context.InitializeSession(url, tenant, company, AuthenticationScheme.UserNamePassword,
navUsername, navPassword);
context.OpenSession();
```

Client Interactions

ActivateControlInteraction(ClientLogicalControl logicalControl)

This interaction resembles the user activating a control, either by tabbing to the control or clicking the mouse inside the control.

SaveValueInteraction(ClientLogicalControl logicalControl, object value)

This interaction resembles the user entering a value in a control and submitting the value to the server.

InvokeActionInteraction(ClientLogicalControl logicalControl)

This interaction resembles the user clicking an action.

OpenFormInteraction

This interaction doesn't really exist as a user interaction. The interaction causes a page of a specific number to be opened.

The Interaction class has 3 properties: Page number, Bookmark, and LoadData.

CloseFormInteraction(ClientLogicalForm logicalForm)

This interaction resembles the user closing a page.

ExecuteFilterInteraction(ClientLogicalControl logicalControl)

This interaction resembles the user activating a quick filter. The interaction has two properties: QuickFilterColumnId and QuickFilterValue.

A number of extension methods have been created on LogicalControl, which gives you the opportunity to invoke or activate a Logical Control directly.

"Catching" pages

Whenever a user interaction causes a page to open, the page will raise an event on the session. You will find a couple of helper methods which will help you setup event handlers and catch a page that you are expecting for a specific interaction.

An example of this is as follows:

```
var page = context.CatchForm(delegate
{
    context.InvokeInteraction(new OpenFormInteraction { Page = "9006" });
});
```

Close NAV session

```
context.CloseSession();
```

Your first end2end example

The following code sample is the simplest end2end example of a user opening a client using Client Services:

```
// Create Session
var userContext = new UserSession.UserContext();
userContext.InitializeSession(url, null, null, AuthenticationScheme.UserNamePassword, navUsername,
navPassword);
userContext.OpenSession();

// Open Role Center
userContext.RoleCenterPage = userContext.CatchForm(delegate
{
    userContext.InvokeInteraction(new OpenFormInteraction { Page = "9006" });
});

// Close Session
userContext.CloseSession();
```

Sample Scenario: Creating a sales order

The following code snippet will create a sales order for customer 10000 with a few lines, all including item 1000. This snippet shows a lot of the things you will need for creating scenarios.

```
public void RunCreateAndPostSalesOrder(UserContext userContext)
{
    // Invoke using the new sales order action on Role Center
    var newSalesOrderPage = userContext.EnsurePage(SalesOrderPageId,
userContext.RoleCenterPage.Action("Sales Order").InvokeCatchForm());

    // Start in the No. field
    newSalesOrderPage.Control("No.").Activate();

    // Navigate to Sell-to Customer No. field in order to create record
    newSalesOrderPage.Control("Sell-to Customer No.").Activate();

    var newSalesOrderNo = newSalesOrderPage.Control("No.").StringValue;
    TestContext.WriteLine("Created Sales Order No. {0}", newSalesOrderNo);

    // Set Sell-to Customer No. to a Random Customer and ignore any credit warning
    TestScenario.SaveValueAndIgnoreWarning(TestContext, userContext,
newSalesOrderPage.Control("Sell-to Customer No."), "10000");
    TestScenario.SaveValueWithDelay(newSalesOrderPage.Control("External Document No."), "10000");

    userContext.ValidateForm(newSalesOrderPage);

    // Add a random number of lines between 2 and 5
    int noOfLines = SafeRandom.GetRandomNext(2, 6);
    for (int line = 0; line < noOfLines; line++)
    {
        AddSalesOrderLine(userContext, newSalesOrderPage, line);
    }
    // Check Validation errors
    userContext.ValidateForm(newSalesOrderPage);

    PostSalesOrder(userContext, newSalesOrderPage);

    // Close the page
    TestScenario.ClosePage(TestContext, userContext, newSalesOrderPage);
}

private void PostSalesOrder(UserContext userContext, ClientLogicalForm newSalesOrderPage)
{
    ClientLogicalForm postConfirmationDialog;
    using (new TestTransaction(TestContext, "Post"))
    {
        postConfirmationDialog = newSalesOrderPage.Action("Post...").InvokeCatchDialog();
    }

    if (postConfirmationDialog == null)
    {
        userContext.ValidateForm(newSalesOrderPage);
        Assert.Inconclusive("Post dialog can't be found");
    }

    using (new TestTransaction(TestContext, "ConfirmShipAndInvoice"))
    {
        ClientLogicalForm dialog =
userContext.CatchDialog(postConfirmationDialog.Action("OK").Invoke);
        if (dialog != null)
        {
            // after confirming the post we don't expect more dialogs
            Assert.Fail("Unexpected Dialog on Post - Caption: {0} Message: {1}", dialog.Caption,
dialog.FindMessage());
        }
    }
}

private void AddSalesOrderLine(UserContext userContext, ClientLogicalForm newSalesOrderPage, int
line)
{
    // Get Line
    var itemsLine = newSalesOrderPage.Repeater().DefaultViewport[line];

    // Activate Type field
    itemsLine.Control("Type").Activate();

    // set Type = Item
    TestScenario.SaveValueWithDelay(itemsLine.Control("Type"), "Item");

    // Set Item No.
    TestScenario.SaveValueWithDelay(itemsLine.Control("No."), "1000");

    string qtyToOrder = SafeRandom.GetRandomNext(1, 10).ToString(CultureInfo.InvariantCulture);

    TestScenario.SaveValueAndIgnoreWarning(TestContext, userContext, itemsLine.Control("Quantity"),
qtyToOrder);

    TestScenario.SaveValueAndIgnoreWarning(TestContext, userContext, itemsLine.Control("Qty. to
Ship"), qtyToOrder, "OK");
```

```
    // Look at the line for 1 seconds.
    DelayTiming.SleepDelay(DelayTiming.ThinkDelay);
}
```

Selecting customers and items randomly

When looking at the code above, you quickly discover that all sales orders are created for customer 10000 and the items we use is always item 1000. This will of course not give a true picture if how the system is used.

You will need to build a mechanism for selecting customers and items more intelligently.

My recommendation is to use OData for this. You would need to expose the Customer card and the Item card as web services, create a service reference to the OData base endpoint, and then create functions like this (this samples returns a customer with Location_Code == Yellow).

You could also build into the scenario that the user uses the drop down to pick a customer from the list or he could search for the customer. This is what the sample projects are using.

It really depends on what the users are doing and, in real life, they will probably know the customer number in xx% of the time. In yy% of the time, the user will be using filtering to find the customer and, in some cases, he might be browsing a list for the customer. You could choose to build this behavior into the scenario.

```
private static object customersYellowLockObj = new object();
private static Dictionary<string, List<string>> _customersYellow = new Dictionary<string,
List<string>>();

/// <summary>
/// Get Random Customer No with Location Code Yellow
/// </summary>
/// <param name="tenant">Tenant to use or null if single tenancy</param>
/// <returns>a Customer No</returns>
private static string GetRandomCustomerNoWithLocationCodeYellow(string tenant)
{
    var rnd = new System.Random();
    if (tenant == null)
        tenant = string.Empty;
    lock (customersYellowLockObj)
    {
        List<string> _cust;
        if (!_customersYellow.TryGetValue(tenant, out _cust))
        {
            string username = string.IsNullOrEmpty(tenant) ? navUsername : tenant + @"\" +
navUsername;

            var service = new NAV.NAV(new Uri(navODataUrl));
            service.Credentials = new System.Net.NetworkCredential(username, navPassword);
            service.IgnoreResourceNotFoundException = true;
            var customersQuery = from customer in (string.IsNullOrEmpty(tenant) ?
service.CustomerCard : service.CustomerCard.AddQueryOption("tenant", tenant))
                                 where customer.Location_Code.Equals("YELLOW")
                                 select customer;
            _cust = new List<string>();
            foreach (var customer in customersQuery)
                _cust.Add(customer.No);
            _customersYellow.Add(tenant, _cust);
        }
        return _cust[rnd.Next(0, _cust.Count)];
    }
}
```

This method supports both single and multi-tenancy. If your tests are designed only for single or multi-tenancy, then you can, of course, remove that complexity.

Test

In order to run the above scenario in the test runner or as a performance test, you need to create a TestMethod which invokes the scenario. As you can imagine, we cannot create a user session for every single test method. This would not resemble the actual system usage from users.

For this, there is a class called a UserContextManager. The UserContextManager can keep a queue of UserContexts and reuse those on request. For every user type, you will create a function which instantiates a UserContextManager:

```
private UserContextManager CreateUserContextManager()
{
    if (UseWindowsAuthentication)
    {
        // Use the current windows user
        orderProcessorUserContextManager = new WindowsUserContextManager(
                NAVClientService,
                null,
                null,
                OrderProcessorRoleCenterId);
    }
    else
    {
        // Use Username / Password authentication
        orderProcessorUserContextManager = new NAVUserContextManager(
               NAVClientService,
               null,
               null,
               OrderProcessorRoleCenterId,
               NAVUserName,
               NAVPassword);
    }
    return orderProcessorUserContextManager;
}
```

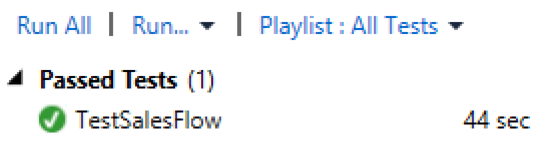
This function will create a UserContextManager, which can create UserContext's for our scenarios and return them.

The UserContextManager has a number of properties for determining the tenant, company, and RoleCenter of the users that are created. If you want to distribute users over a series of random tenants, then you will need to override the CreateUserContext method and create the UserContext in that method.

With this in place, your TestMethod could look like this:

```
[TestMethod]
public void CreateAndPostSalesOrder()
{
    TestScenario.Run(OrderProcessorUserContextManager, TestContext, RunCreateAndPostSalesOrder);
}
```

With this in place, you can right click the TestMethod and invoke Run or Debug Tests. The Test Explorer will open and reveal the result of the test:



```
◢ Passed Tests (1)
  ✓ TestSalesFlow            44 sec
```

All your other test methods will also be available from the Test Explorer to run and debug. When all tests runs successfully, it is time to start thinking about performance testing.

The code for distributing users over multiple tenants or companies would need to be inserted to get tenant names from a collection or the like.

Execute

After successfully writing and executing scenario tests, it is time to run these tests as performance tests.

Please read this MSDN article, which talks generally about performance testing:
http://msdn.microsoft.com/library/dn250793.aspx

The actual test methods used in this article are http recordings for a web site. In our case, we are not recording any web traffic. When running Microsoft Dynamics NAV performance tests in Visual Studio, we will be creating our user scenarios as test methods in C#, and then create a test mix by indicating the average frequency of this user scenario per user per hour.

You will not be creating user types as such, but based on the table from earlier with usage types and user scenario frequencies, you should be able to create a test mix, which resembles the average user load.

Let’s say you have these two user types.

- User1
    - Sales order twice per hour
- User2
    - Purchase order once a day

And, you typically have 4 User1’s per each User2.

Then, your average user would be creating 8/5 = 1.6 sales orders per hour and 1/8/5 = 0.025 purchase orders per hour.

When we later ask Visual Studio to simulate 400 users – then the sales order test method will be executed 640 times an hour, and the purchase order test method will be executed 10 times an hour. It is then up to the test method to ensure that we spin up the right number of sessions on Microsoft Dynamics NAV.

If the users are Windows client users, who open Microsoft Dynamics NAV every morning and keep it open throughout the day – then you actually want to ensure that you spin up 400 user sessions on the Microsoft Dynamics NAV Service tier and distribute the load between these.

If the users are Web client or Portal users, only connecting and using the system when needed, then you just want to spin up the number of Microsoft Dynamics NAV sessions on the service tier, which actually would be necessary at any given time.

In the last section, I described the ContextManager. It is important to understand, that when you want to simulate a large number of users, you might need more than one machine to do this. Visual Studio performance test supports distributing the work to a number of Visual Studio test agents.

Creating a test agent is really simple – create a Windows Server computer in the same domain as your test controller (development box setup as test controller). Then install Visual Studio test agent on the computer and register the computer in your test controller.

The ContextManager will run on every agent. This means that if you are simulating 100 order processors, you would want to distribute those over the number of available agents.

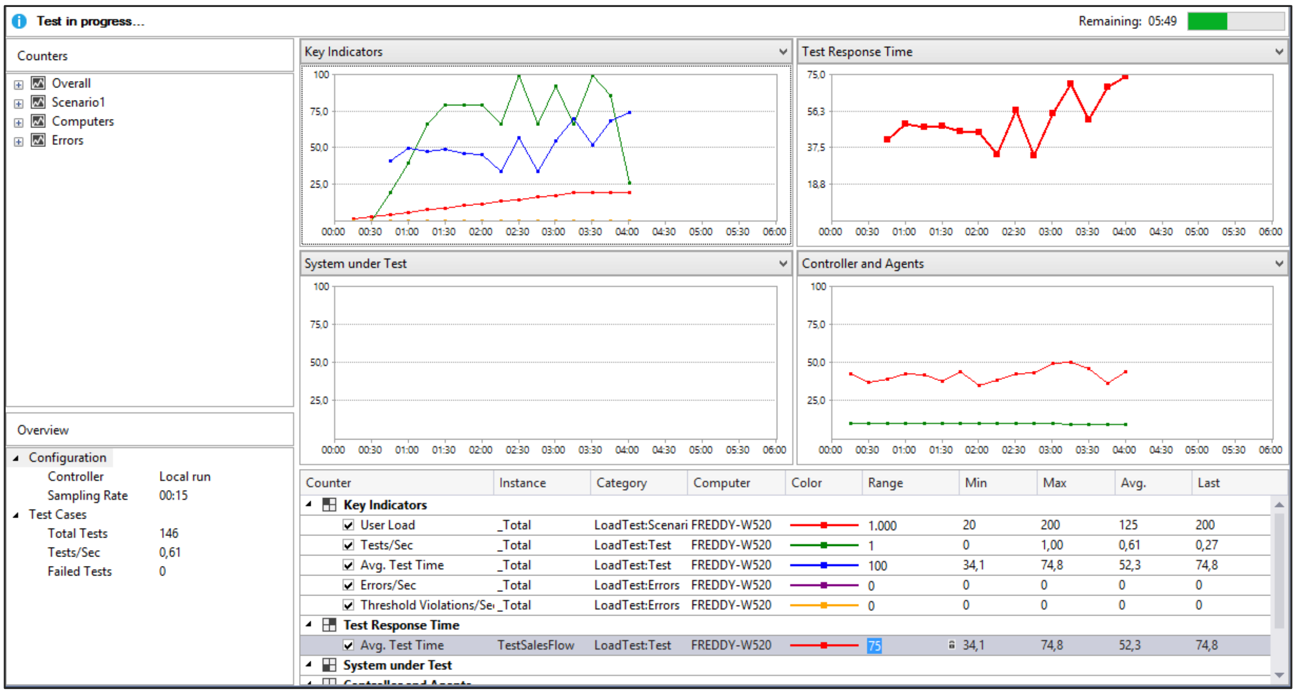
If you are simulating Windows client users, you should keep the server connection between scenarios. It is important to make sure that you spin up more user contexts than what you actually need. This can be done by implementing a queue of UserContexts, which you then use in turn. The number of UserContexts in this queue should be the total number of users divided by the number of agents.

Having defined the performance tests, there are several ways to run them.

1. From your local computer.
2. Using a Visual Studio perf controller and a number of agents.
3. Using Visual Studio Online.

Which one you select is determined by the TestSettings of your solution.

Performance tests executing through local computer:

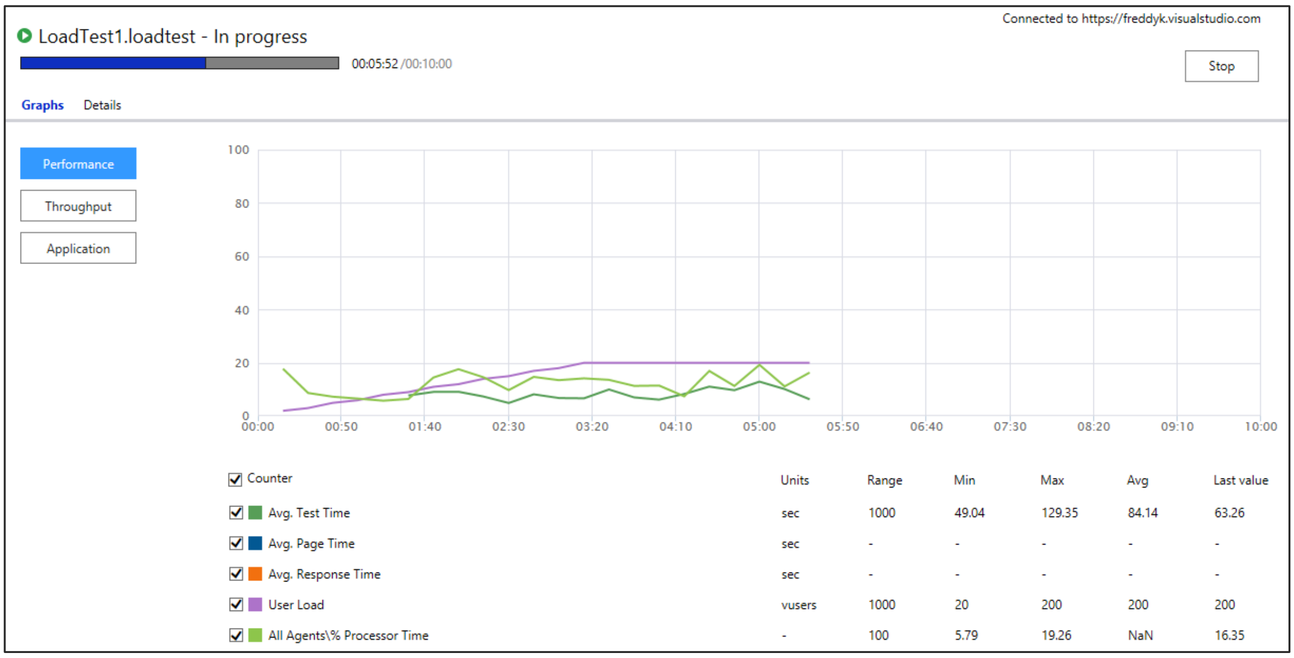


Overview

Configuration	
Controller	Local run
Sampling Rate	00:15
Test Cases	
Total Tests	146
Tests/Sec	0,61
Failed Tests	0

Counter	Instance	Category	Computer	Color	Range	Min	Max	Avg.	Last
Key Indicators									
✓ User Load	_Total	LoadTest:Scenari	FREDDY-W520	🟥	1.000	20	200	125	200
✓ Tests/Sec	_Total	LoadTest:Test	FREDDY-W520	🟩	1	0	1,00	0,61	0,27
✓ Avg. Test Time	_Total	LoadTest:Test	FREDDY-W520	🟦	100	34,1	74,8	52,3	74,8
✓ Errors/Sec	_Total	LoadTest:Errors	FREDDY-W520	🟪	0	0	0	0	0
✓ Threshold Violations/Se	_Total	LoadTest:Errors	FREDDY-W520	🟧	0	0	0	0	0
Test Response Time									
✓ Avg. Test Time	TestSalesFlow	LoadTest:Test	FREDDY-W520	🟥	75	34,1	74,8	52,3	74,8
System under Test									

Performance tests executing through Visual Studio Online:


LoadTest1.loadtest - In progress
00:05:52 /00:10:00
Connected to https://freddyk.visualstudio.com
Stop
Graphs
Details
Performance
Throughput
Application

Counter	Units	Range	Min	Max	Avg	Last value
✓ 🟩 Avg. Test Time	sec	1000	49.04	129.35	84.14	63.26
✓ 🟦 Avg. Page Time	sec	-	-	-	-	-
✓ 🟧 Avg. Response Time	sec	-	-	-	-	-
✓ 🟪 User Load	vusers	1000	20	200	200	200
✓ 🟩 All Agents\% Processor Time	-	100	5.79	19.26	NaN	16.35

When the performance tests completes, you need to analyze the test results.

Testing on multiple servers

If you want to run Microsoft Dynamics NAV as a server farm (multiple Microsoft Dynamics NAV Service tiers behind a load balancer), I recommend that you read the whitepaper about Large Scale Hosting on Windows Azure – even if you are not using Windows Azure.

If you are running multiple Microsoft Dynamics NAV Service tiers behind a load balancer, you have probably set up a simple redirect site or you are using Application Request Routing (ARR).

If you are running ARR, it might not give the right picture of the load, because ARR works by sending a cookie to the client (in our case, the agent) with information about which server to communicate.

If you are running multiple servers, then every agent will get a cookie, and you might end up with an uneven distribution of load.

So if you are running ARR – you would want to open individual ports for each web client and create a redirect site – or make a round robin load balancer mechanism in the CreateUserContext method.

Analyze

The analyze phase consists of two phases:

- Monitoring the performance tests while they are running
- Analyzing results after running performance tests

As stated in the section about the performance testing process, you will probably find yourself chasing bottlenecks until you are satisfied with the numbers you get.

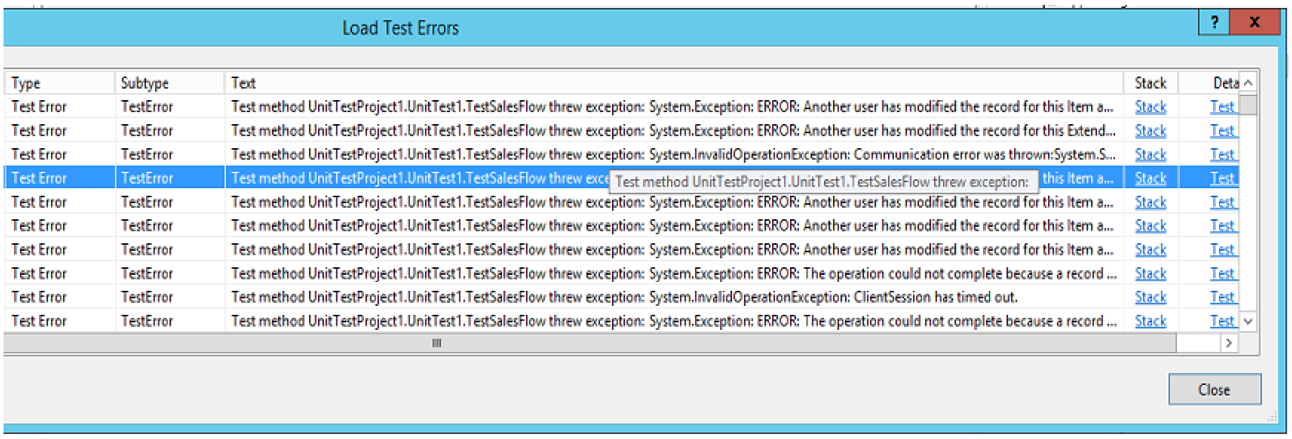
Some of these bottlenecks will be configuration changes and some might be application changes.

Track errors in the performance test scenarios

When writing performance test scenarios, it is important to take normal business challenges into account. This includes situations like out of stock warnings, item variants, and other situations where pages pop up when a well-defined situation occurs.

Locking

If the performance tests cause locking issues, it will cause exceptions to occur in the scenario code, and as a result, the tests will fail and you will be able to see the reason for the failing test in the list of Performance Test Errors:


Load Test Errors

Type	Subtype	Text	Stack	Deta
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: Another user has modified the record for this Item a...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: Another user has modified the record for this Extend...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.InvalidOperationException: Communication error was thrown:System.S...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exce Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: this Item a...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: Another user has modified the record for this Item a...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: Another user has modified the record for this Item a...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: Another user has modified the record for this Item a...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: The operation could not complete because a record ...	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.InvalidOperationException: ClientSession has timed out.	Stack	Test
Test Error	TestError	Test method UnitTestProject1.UnitTest1.TestSalesFlow threw exception: System.Exception: ERROR: The operation could not complete because a record ...	Stack	Test

There might not be anything wrong with your tests if this happens. It might be caused by the way your application is built, and you will likely see these issues when running Microsoft Dynamics NAV with real users as well.

Test agents trending towards 100%

When running the performance tests, you can see how your agents are doing. If the agents are running out of memory or out of CPU, then you probably just need to add more test agents. Your numbers will not be correct if you keep adding users on exhausted test agents, as these will run at a lower pace and in the end, probably won't put the required load on the system.

Microsoft Dynamics NAV Servers trending towards 100%

Depending on the scenarios, every user will take up between 5 and 20Mb of RAM on the Microsoft Dynamics NAV Service tier and between 5 and 20Mb of RAM on IIS.

If users are online throughout the day, the performance tests should keep the sessions alive on the Microsoft Dynamics NAV Server and the server should have sufficient RAM to handle this.

The CPU utilization will still only be running the actual scenarios, and the idle user sessions will just be occupying RAM.

If the Microsoft Dynamics NAV Server gets pressed on memory, you might also see the server spike in CPU usage due to the .NET memory garbage collection. The performance counters on the Microsoft Dynamics NAV Server which could reveal this are:

```
\.NET CLR Memory(Microsoft.Dynamics.Nav.Server)\% Time in GC
\.NET CLR Memory(w3wp)\% Time in GC
```

SQL Server trending towards 100%

Depending on the scenarios, every user will put some load on the SQL Server when performing scenarios.

There are multiple reasons why SQL Server could trend towards 100%. You could have non-optimal code in Microsoft Dynamics NAV, which causes too many SQL calls.

It is also possible that your data or metadata cache settings are set too low.

If you are running multiple tenants, your data cache is shared between tenants and might need to be increased. Performance counters which could indicate this problem are:

```
\Microsoft Dynamics NAV(nav)\% Primary key cache hit rate
\Microsoft Dynamics NAV(nav)\% Result set cache hit rate
```

The primary key cache hit rate should be above 90%; otherwise it indicates that your cache might be too small.

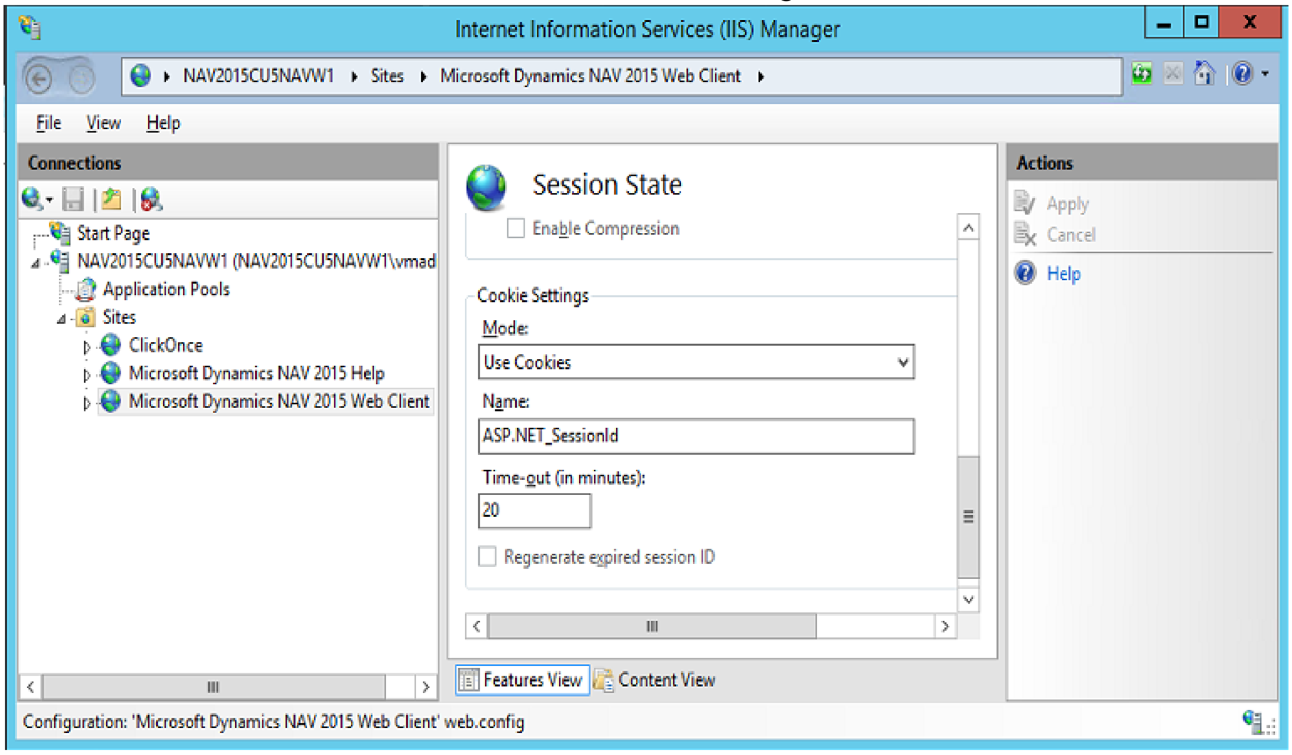
Sessions times out

If you experience sessions timing out, you might need to change the timeout settings on IIS and in the web.config file. A couple of settings which you can change in order to modify the timeout are described as follows:

The outermost web.config file (located in c:\inetpub\wwwroot\<instance>) has a sessionTimeout setting for controlling the timeout for session from the Microsoft Dynamics NAV Service tier. This is the maximum time that the server will keep the session alive.

```
<add key="SessionTimeout" value="00:20:00" />
```

The IIS Manager Session State settings for the Web client controls the timeout for the session cookie. This should be the same or lower than the SessionTimeout setting.



Internet Information Services (IIS) Manager
NAV2015CU5NAVW1 ▸ Sites ▸ Microsoft Dynamics NAV 2015 Web Client ▸
File
View
Help
Connections
Start Page
NAV2015CU5NAVW1 (NAV2015CU5NAVW1\vmad
Application Pools
Sites
ClickOnce
Microsoft Dynamics NAV 2015 Help
Microsoft Dynamics NAV 2015 Web Client
Session State
Enable Compression
Cookie Settings
Actions
Apply
Cancel
Help
Regenerate expired session ID
Features View
Content View

Mode: Use Cookies
Name: ASP.NET_SessionId
Time-out (in minutes): 20

Configuration: 'Microsoft Dynamics NAV 2015 Web Client' web.config

The innermost web.config file (located in c:\inetpub\wwwroot\<instance>\WebClient) has a timeout on the forms node when using forms based authentication.

```
<forms loginUrl="~/SignIn.aspx" timeout="2880" />
```

The Timeout setting here is the number of minutes until your authentication will expire. This value should be set higher than your SessionTimeout setting in IIS.

Microsoft Dynamics NAV Server cannot accept more connections

You need to increase the maximum number of concurrent connections in CustomSettings.config file. Look at *ClientServicesMax* Concurrent Connections setting.

Modify

Application

If the results of the performance tests don't meet expectations, you always need to consider whether code in the application is part of the reason.

You might need to investigate your code and refactor based on your findings during the execution of performance tests on your solution.

Number of Microsoft Dynamics NAV Service Tiers

When a client connects to Microsoft Dynamics NAV Server, the load balancer will select a server and keep the connection open throughout the lifetime of the client. This means that setting up more servers behind a load balancer will increase the capacity of the cluster of Microsoft Dynamics NAV Service tiers.

Server Memory

As a rule of thumb, each Microsoft Dynamics NAV Service tier needs 500Mb of memory to run.

On top of that, the Microsoft Dynamics NAV Service tier needs memory for each active session, even if they are idle. The more pages the user have opened, the more memory gets allocated, but a good starting point for doing sizing calculations is around 10Mb per active session.

Idle sessions are not terminated if they are running the Windows client. When running the Web client, idle sessions will be terminated after 20 minutes (this is configurable).

Server CPU (cores and speed)

Faster CPU will get things done faster, and more cores can run more things in parallel. Idle sessions on the Microsoft Dynamics NAV Service tier don't use any CPU power, except for generic housekeeping, which shouldn't be counted.

Active sessions will use CPU power depending on what they do, but it is hard to generalize how much. For example, CPU power is used whenever the user is performing a task that requires the Microsoft Dynamics NAV Service tier to run some business logic or perform some UI logic or setup server pages. If the Microsoft Dynamics NAV Service tier does a lot of IO (SQL calls or client callbacks), then the CPU will be waiting for response and thus be loaded less.

Data Cache Size

This is a Microsoft Dynamics NAV Server setting, which located in CustomSettings.config file.

Sets the data cache size. This is an abstract value with contextual meaning on the type of the item being cached.

```
<add key="DataCacheSize" value="9" />
```

The number you specify in DataCacheSize setting determines how much memory is used for caching data. The actual amount of memory (in MB) allocated is 2^n, where n is the value of the DataCacheSize setting:

Value	Memory
9 (default)	512Mb
10	1Gb
11	2Gb
12	4Gb
13	8Gb
14	16Gb
15	32Gb
…	

When running a single tenant system the default value of 9 is probably good.

When running a multi-tenant system, the data cache is shared between all tenants. When the cache is full (LRU mechanism), the oldest object in the queue will be disposed when a new entry is added to the cache.

If you have 512 tenants with equal usage distribution and only 512Mb data cache, you will probably end up utilizing the cache very poorly.

As described in the monitor section, you can monitor the Primary Key Cache Hit Rate and the Result Set Cache Hit Rate to determine whether or not you need to increase the data cache.

Metadata Provider Cache Size

This is a Microsoft Dynamics NAV Server setting, which located in CustomSettings.config file.

Sets the Metadata Provider cache size (in number in objects cached). Set to 0 to disable cache.

```
<add key="MetadataProviderCacheSize" value="150" />
```

You can monitor the number of objects in the metadata cache by monitoring the following performance counter on the Microsoft Dynamics NAV Service tier. There is really no reason not to set this to a high value (like 10000) if you are not pressed on memory on the Microsoft Dynamics NAV Server.

Max Concurrent Calls

This is a Microsoft Dynamics NAV Server setting, which is located in CustomSettings.config file.

Maximum number of concurrent client calls that can be active on the Microsoft Dynamics NAV Server. To disable this setting set the value to "MaxValue".

```
<add key="MaxConcurrentCalls" value="40" />
```

The number specified here determines how many concurrent calls the Service Tier is able to handle. The more cores in your server, the higher this value can be.

If the number of attempted calls to the server exceeds this number, users will be placed in queue and will wait for the server to be ready. The call won't fail unless a timeout is met.

If this value is set too high, you might overload your server with poor performance as a result.

Max Concurrent Connections

This is a Microsoft Dynamics NAV Server setting, which is located in CustomSettings.config file.

The maximum number of concurrent client connection that the service will accept. To disable this setting set the value to "MaxValue".

```
<add key="ClientServicesMaxConcurrentConnections" value="150" />
```

The number specified here determines the number of active sessions you can have on the Microsoft Dynamics NAV Service tier. A value of 150 means that user number 151 who tries to connect will get an error.

You can monitor the number of concurrent connections by monitoring the following performance counter on the Microsoft Dynamics NAV Service tier:

```
\Microsoft Dynamics NAV(nav)\# Active Sessions
```

The performance counter called Open Connections shows the number of SQL connections Microsoft Dynamics NAV has open. This number will be slightly higher than the number of Active Sessions.

More information

Below you will find a number of links to places where you can find more information.

Please note that the list is not exhaustive as more things will be available over time.

How-to videos

How Do I: Run NAV Performance Tests Using Visual Studio in Microsoft Dynamics NAV
This video demonstrates how to run Microsoft Dynamics NAV performance test by using Visual Studio

How Do I: Write Microsoft Dynamics NAV Load Tests Scenarios Using Visual Studio: Part 1
This video demonstrates how to write Microsoft Dynamics NAV load tests scenarios using Visual Studio

How Do I: Write Microsoft Dynamics NAV Load Tests Scenarios Using Visual Studio: Part 2
This video demonstrates how to write Microsoft Dynamics NAV load tests scenarios using Visual Studio

Blog's

PartnerSource
PartnerSource Get Ready page.

http://blogs.msdn.com/b/nav/
The Microsoft Dynamics NAV Team Blog

https://navperformance.wordpress.com/
Blog about Performance Testing

GitHub organizations

https://github.com/NAVPERF
GitHub organization for performance testing Microsoft Dynamics NAV

+ Share

Microsoft Development Center Copenhagen
Frydenlunds Alle 6
2950 Vedbaek
Denmark