Design Patterns for Distributed Systems

Implementing GraphQL APIs
VUE.js for jQuery Developers
Azure Machine Learning
Features

8 Azure Machine Learning Workspace and MLOps
It’s when you’re working with lots of data that you start looking around for an easier way to keep track of it all. Machine learning and artificial intelligence are the obvious answers, and Sahil shows you why.
Sahil Malik

16 A Design Pattern for Building WPF Business Apps: Part 3
In the third installment of his WPF series, Paul shows you how to get feedback using an Entity Framework entity class. He also shows you how to start expanding your activities, like adding, editing, or deleting screens.
Paul D. Sheriff

24 Responsible Package Management in Visual Studio
If you use a package management tool, like NuGet, Node Package Manager (NPM) for JavaScript, or Maven for Java, you already know how they simplify and automate library consumption. John shows you how to make sure that the packages you download don’t cause more troubles than they solve.
John V. Peterson

30 Moving from jQuery to Vue
Even if you don’t need the enormity of a SPA, you don’t have to lose the benefits of a framework. Shawn recommends using Vue to simplify the code and make it both more reliable and more testable.
Shawn Wildermuth

36 Intro to GraphiQL for .NET Developers: Schema, Resolver, and Query Language
Peter introduces you to GraphQL so your REST API client list can grow and change without a lot of pain. You can use strongly-typed schema, eliminate over- and under-fetching, and you can get analytics about how clients are really using your API.
Peter Mbanugo

42 Design Patterns for Distributed Systems
Stefano explores using containers for reusable components and patterns to simplify making reliable distributed systems. He teases on microservices to place all functionality within a single application.
Stefano Tempesta

46 Nest.js Step-by-Step: Part 2
Bilal continues showing us just how interesting, useful, and easy it is to integrate Nest.js with TypeORM. You’ll get to replace mock data from the first article with real data this time, too.
Bilal Haidar

54 Cross-Platform Mobile Development Using Flutter
Using Flutter, Google’s latest cross-platform framework for developing iOS and Android apps, Wei-Meng shows you how easy developing mobile apps can be.
Wei-Meng Lee

70 Add File Storage to Azure App Services: The Work Around
When maintaining the hierarchy of a file system and integrating security limits you to a single point of access, you might have some heavy lifting to do while you wait for Microsoft to supply a tool to automate this task. Mike and his team found a great work-around that will keep you happy until the tool is available.
Mike Vaagner

Columns

78 Managed Coder: On Time
Ted Neward

Departments

16 Editorial
18 Advertisers Index
72 Code Compilers
Code Smells Are Universal

Over the years, I’ve become fluent in several programming languages: C#, JavaScript, Visual Basic.NET, Ruby, FoxPro, and a few others. Last month, I started the process of adding Python to my repertoire because my development team is currently in the process of building a data processing platform.

This platform pulls data from multiple sources of data and uses Python (with its rich ecosystem of statistical libraries) to run various models over the data. I was tasked with integrating these Python modules into our ETL pipeline, so I asked the data analyst for a copy of the code to determine first, how it works and second, how I was going to integrate this code into our pipeline.

I spent some time with the developer. The smell of the code became apparent rather quickly. When developing the code, the analyst implemented a metadata-driven approach to loading and running modules for each client. The application loaded up the client code and used the parameters attached to that client to make it simple to maintain.

At first blush, this was a good sign. This code “smells” rather nice. Upon further digging, I found some code that has a distinctly unpleasant odor. The main program accepted a number of dynamic command arguments. These parameters read and assigned to different memory variables. Okay, so far so good. Where was the smell? The smell came from a called module that read the command-line arguments:

```python
start_date = “%Y-%m-%d”
send_date = “%Y-%m-%d”
```

It didn’t look correct to me. It shouldn’t be the job of the called program to read the command-line parameters from the calling modules argument list. This was a definite smell to me.

I know that THIS is not an interesting story. The interesting part is that I was able to identify a code smell in an unfamiliar programming language. You see: Code Smells are Universal. Let’s take a look at some JavaScript code used to validate the format of a date string in Figure 1. For reference, the correct format of the string is as follows: 1977-05-25 01:30:30

This code has several different smells. First, it has a bit of stinker code in that it uses a force to validate a date time string. Can you think of better ways to write this validation? The first idea that comes to mind is that this code could probably be handled by a regular expression. So, does this code have a bad or a good smell?

When it comes to code, whether it has a good or bad smell is a subjective thing. This code is probably a mix of both. The bad smell comes from its brutish nature. It basically validates each character one at a time. The good part is the intention of the code; when an error does occur, the code tells the user EXACTLY what’s wrong with the time string.

Finally, other smells can be determined by answering the following questions:

- Does the code work as designed?
- Is the code maintainable?
- Is the code understandable?

In my judgement, the answers to these questions for this bit of code is yes. Even if you don’t write a lot of JavaScript code, you can decide for yourself whether the code is any good or not. What comments would you make about this code? Tell you what, Ping me at @odspedlock on Twitter. I’d love to hear your comments about this code, good or bad. Please be kind though.

After spending some time thinking about the Python code, I came to the realization that most programming falls back on the old premier. It’s the concept that matters. By spending time mastering concepts, I’ve been able to master multiple languages. And now I’ve also found a new superpower: the ability to look at code in unfamiliar languages and determine whether or not it has code smell, both good and bad.

Why AceThinker Screen Grabber Pro?

- **Record all desktop activities**: Equipped with different recording modes, AceThinker Screen Grabber Pro can record the entire screen area, a specific area, an application window, and more. Aside from the desktop screen, the tool can also capture audio from the system and microphones simultaneously. This is essential for people who make instructional videos as they can incorporate audio directly onto the video.

- **Create scheduled tasks**: The tool has a task scheduler option that enables the users to set a specific time to record automatically. This is an efficient way to record live-streams, webinars, or the internet activity of your kids, and to schedule regular recordings even if you’re not around.

- **Edit video during and after recording**: Annotate while recording with the built-in editing panel of the tool. There are various video enhancement options available that can be added as the recording progresses. This enables you to process the video easily and saves a lot of time and effort in post-editing.

- **Save and share screencast**: After recording the video, you can convert the recorded videos into desired formats for watching on various devices. You can also upload them to a cloud server or share your videos on websites like YouTube and more.

About AceThinker Software

AceThinker Limited was established in 2010 and continues to provide digital multimedia solutions to many households and businesses. Over the years, AceThinker Limited steadily gained popularity by releasing essential multimedia tools that provide different solutions to various situations. AceThinker Screen Grabber Pro is the premiere offering of AceThinker Limited since its launch. To learn more about the software, please visit https://acethinker.com/desktop-recorder or scan the QR code with your smart phone.
In my previous article (https://www.codemag.com/Article/1907021/Azure-Machine-Learning-Service), I discussed the Azure Machine Learning Service. The Azure Machine Learning Service is at the core of custom AI. But what really ties it together is the Azure Machine Learning workspace. The process of AI involves working with lots of data, cleaning the data, writing and running experiments, publishing models, and finally collecting real-world data and improving your models. The machine learning workspace provides you and your co-workers with a collaborative environment where you can manage every aspect of your AI projects. You can also use role-based security to define roles within your teams, you can check historical runs, versions, logs etc., and you can even be to your Azure DevOps repos and fully automate this process via ML Ops.

In this article, I’ll introduce you to all of these and more.

Provision an ML Workspace
Creating an ML workspace is extremely easy. Log into portal.azure.com using an account with a valid Azure subscription, search for Machine Learning Service Workspace, and click on the Create button in the provided blade. You’ll be asked to provide a name; for the purposes of this article, choose to create it in a new resource group. The names I picked were sailhwsworkspace for the name of the workspace and ML for the name of the resource group. And in just about a minute or so, your Azure Machine Learning service is created.

You may also create an Azure Machine Learning service workspace using the Azure CLI. In order to do so, you first must install the Azure CLI machine learning extension using the command:

`az extension add -n azure-cli-ml`

You can then create an Azure Machine Learning workspace like this:

```
az group create --name ml --exists
az ml workspace create --name sailhwsworkspace --rg ML
```

Once the workspace is created, you’ll notice a number of newly created resources in your subscription, as can be seen in Figure 1.

As you can see from Figure 1, the Azure Machine Learning workspace depends on a number of other services in Azure.

Walkthrough of the ML Workspace
At this time, you’ve only created a workspace; you haven’t yet put anything in it. So before you go much further, let’s examine the major components of the ML workspace. I won’t dive into every single aspect here, but just focus on the interesting major players. Go ahead and visit the workspace. Within the workspace you should see a section like that shown in Figure 2.

As can be seen in Figure 2, the Activity Log is a great place to learn what activities have been performed in the workspace. Remember, you’re not the only one using this workspace; it’s a collaborative area that you share with your co-workers. When an experiment goes away and starts giving out awful results, this is where you can go and find out exactly what happened recently.

Remember, AI projects need to be secured just like any other project. Perhaps even more so, because as we move forward in time, we will rely more, not less, on AI. In fact, AI systems will be used to hack non-AI systems, such as your friendly local powerplant. It’s crucial that you know and preserve a history of activities going on in your environment.

The second interesting thing you see here is the Access Control (IAM) section. Azure Machine Learning workspace relies on the usual Azure Identity and Access Management (IAM) to secure resources and provide resources. You can define your own roles as well, but the Azure Machine Learning workspace comes with numerous useful prebuilt roles. For instance, you don’t want just anyone to deploy a model, right? Additionally, perhaps you want the log readers, well, just read—not edit, not even accidentally—the experiment. All of this can be neatly tied down using regular Azure IAM.

Perhaps a superfluous point here is that the Azure Machine Learning workspace is part of the Azure portal. It’s therefore protected by your Azure AD and gains all the benefits of Azure AD, such as MFA, advanced threat protection, integration with your corporate on-premises identities, etc.

The Azure Machine Learning workspace is part of the Azure portal and therefore protected by your Azure AD.

Publish and Deploy using Azure CLI
The next important section is the assets section, as can be seen in Figure 3.

This area is where you can view and manage your actual work-your experiments, your models, the compute you provision, etc. To understand this section better, let’s publish and run an experiment and see the entire process end-to-end.

Create a Model
Remember that for the purposes of this article, the actual experiment is unimportant. The same instructions apply to any kind of problem you may be attempting to solve. I’ll use an available open-source database that’s available at https://www.kaggle.com/datasets/sona-ofu/iris-dataset.

This dataset includes ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements that were collected for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline. Using this data, I can create a simple regression model and project the progression of the disease in a patient given the ten baseline variables about the patient. The code for this experiment is really straightforward and can be seen in Listing 1.

The next step to submitting this an experiment as you do can be easily using the portal Azure ML SDK or via the Azure CLI. I’ll show you how to do this using the Azure CLI.

First, attach yourself to the resource group and folder. This command isn’t 100% necessary, but it’ll help by not requiring you to specify the resource group and folder over and over again every time you wish to execute a command.

```
adl ml folder attach -a sailhwsworkspace -g ML
```

Once you’ve run the above command, you can now go ahead and request to have an Azure ML compute resource created for you. Note that a compute resource comes in many shapes and sizes. Here, you’re creating a standard VM compute with one node. You can create this resource using the command:

```
adl ml compute target create -a sailhwsworkspace -p mycomputecluster -n nodes 1 -m nodes 1 -s STANDARD_DS_v2
```

It’s worth pointing out that the ML workspace gives you full control over virtual network settings, so you can keep this compute resource or associated storage accounts etc. in their own virtual network, away from the prying eyes of the Internet. Your InfSec team will probably be happy to hear that their valuable and sensitive training data will always be secure.

Once the above command finishes running, you should see a compute resource provisioned for you, as shown in Figure 4.

The name of the compute resource is important. Now I wish to be able to submit my experiment and in order to submit it, I

```
name = "test" type = "compute" state = "provisioned"
```

You can then submit the experiment as you can see in Figure 5.
need to supply a configuration. This configuration file resides in the .azureml folder in a file called sklean.unconfig. You can see my sklean.unconfig in Listing 2. Of special note in Listing 2, is the value of "target". Look familiar? That's the name of the compute target you created earlier.

You also need to provide the necessary dependencies your experiment depends on. I've chosen to provide those in a file called training-env.yml, the contents of which can be seen in Listing 3.

Assuming that you have a config.json in your .azureml folder pointing to the requisite subscription and ML workspace, you can submit an experiment using the following command:

```
aml run submit-script
- ! sklean -e test
- ! training-env.yml

train sklean.py
```

By running the above command, you'll get a link to a Web view where you can track the status of the submitted run. At this time, you can just wait for this command to finish, or observe the status of the run under the "Experiments" tab under your ML workspace.

Once the run completes, notice the ML workspace automatically stores a lot of details for the run, as can be seen in Figure 5.

Here are some of the details that the Azure ML workspace automatically keeps a track of for you.

It stores all the runs, along with who initiated them, when it was run, and whether or not it succeeded. It also plots the metrics as charts for you, so you can visually tell the output of a run.

Under the outputs tab, it stores all logs and outputs. The outputs can be the models, for instance. And finally, as you saw in Figure 5, it stores a snapshot of what was run to produce those outputs, so you have a snapshot in time of what you've about to register and deploy next.

Register a Model

In the tabs shown in Figure 5, under the Outputs tab, you can find the created models. Go ahead and download any one of the models, which should be a file ending in .job. The next thing you need to do is use this file and register the model.

In order to register the model, you can use either the ML SDK, Azure CLI, or do it directly through the browser UI. If you choose to do this using Azure CLI, you can simply use the following command:

```
az ml model register -n mymodel -p sklean.regression_model. \\
job -t myml -j job
```

This command relies on three inputs. First is the name of the model you're creating, which is mymodel. The model file itself is sklean.regression_model.job. The model file is a simple JSON file describing the version and workspace for the model. It can be seen here:

```
modelId:"mymodel-2",
```

Model

The following command will deploy your model to an ACI instance:

```
az ml model deploy
-n mymodel
-t myml
--ic inferenceConfig.yml
--dc dcDeployment.yml
--ownerwrite
```

Once you run the above command, you should see an image created for you, as you can see in Figure 7.

In each such created image, you're able to see the specific location on which the Image resides. This is usually an auto-provisioned Azure container registry, and the workspace au-

![Figure 5: Details of the run](image)

![Figure 6: Our newly registered model](image)

![Figure 7: A newly created image](image)
Additionally, you can find a new deployment created for you, as can be seen in Figure 8.

For each deployment, the workspace allows you to track which model the deployment is from and when it was created or updated. This way, you can completely back-track it to which experiment version and dataset the model came from, and who deployed it. At any point, you can choose to update the deployment, and it will track these changes also.

Finally, as you can see in Figure 9, you can grab the scoring URI for your newly deployed model. It’s this scoring URI that your clients can make POST requests to, in order to make predictions against your model.

Automating Using ML Ops

So far in this article, I’ve shown you how to use Azure CLI to run an experiment, create a model, create an image, and deploy a model. In this process, I demonstrated all of the value that Azure Machine Learning workspace adds to the overall process.

But at the center of any AI project is lots of data and algorithms. Data is usually managed in some sort of data store. It could be anything, as long as your code can talk to it. But the brain trust is in the algorithms. The algorithms are written as code, usually Jupyter notebooks. And like any other project, you’ll need to source-control them.

A great way to manage any software project is Azure DevOps. It lets you manage all aspects of a software project. Issues are a big part of DevOps, sprint planning is another, and source control is also an important aspect. A rather interesting aspect of DevOps is pipelines. Pipelines let you automate the process of building and releasing your code via steps. All of these important facets, code, sprints, issues, and pipelines can work together with each other.

An AI project is just like any other software project. It needs code, it needs data, it needs issue tracking, it needs testing, it needs automation. And DevOps can help you automate this entire process, end to end.

As specifically, you can use MLOps to automate everything you’ve seen in this article so far, via a DevOps pipeline. For MLOps to work, there are four main things you need to do.

First, you need to get your code into the DevOps repository. This is not 100% necessary, because DevOps can work with other source control repositories. However, let’s just say that you get your code in some source code repository that DevOps can read from, and because DevOps does come with a pretty good source control repository, perhaps just go ahead and use that.


Once this extension is installed, create a new Azure Resource Manager Service connection, as can be seen in Figure 10.

Provisioning this connection creates a service principal in your Azure tenancy, which has the ability to provision or deprovision resources, as needed, in an automated fashion. It’s this service connection, called ML that is used by the pipeline.

Finally, create a pipeline with the code as shown in Listing 7. Let’s walk through what this pipeline is doing. The first thing you note is that it’s using Azure CLI, and it’s doing so using the service connection you created earlier. Besides that, it’s running on an Ubuntu agent.

It first installs Python 3.6 and then installs all the necessary dependencies that the code depends on. It does so using pip, which is a package installer for Python. Then it adds the Azure CLI ML extensions. This step is necessary because the agent comes with Azure CLI but doesn’t come with ML extensions.

It then attaches itself to the workspace and resource group. This step could be automated further by provisioning and deprovisioning a workspace and resource group as necessary.

It then creates a compute target, followed by running the experiment, registering the model as an image, and creating a deployment, and when you’re done, you delete the compute so you don’t have to pay for it.

All of this is set to trigger automatically if a code change occurs on the master branch.

The end result of all this is that as soon as someone commits code into the master, the whole process runs in an automated fashion, and it creates a scoring URI for you to test.

You get notified of success and failure, and basically all of the other facilities that Azure DevOps offers.
Listing 7: The DevOps pipeline

triggers:
  - master

pool:
  virtimage: 'Ubuntu:16.04'

steps:
  - task: DotNetCoreBuild
    displayName: 'dotnet Core 3.6'
    inputs:
      version: '3.6'
      script:
        - dotnet build
  - task: PublishBuildResults
    condition: succeeded(gracePeriod: 7, failOnDefault: true)
    inputs:
      testResultsFiles: '${{(items('$(Build.SourcesDirectory)\**\*tests\*.xml'))}}'
      testResultsFolder: 'E:\build\Binaries\$(Build.Force生态保护|false)

  - task: AcuraCI
    inputs:
      azureSubscription: 'ML'
      scriptLocation: 'inlineScript'
      inlineScript: 'az ml compute target create -n ml-computetarget -g ml-computetarget -s ml-computetarget -r $env:AGENT_JOB_TYPE -n ml-computetarget -l $env:AGENT_JOB_TYPE -s STANDARD_DS_X2' 
      workingDirectory: 'model-training'

  - task: AcuraCI
    inputs:
      azureSubscription: 'ML'
      scriptLocation: 'inlineScript'
      inlineScript: 'az ml run script -c ml-run-script -i azureml -o test -t training -m train-mlrun -p -w model-training'
      workingDirectory: 'model-training'

  - task: AcuraCI
    inputs:
      azureSubscription: 'ML'
      scriptLocation: 'inlineScript'
      inlineScript: 'az ml model register -n mymodel -g mlcompute_regression_model -t model.json -l mlmodel -d model-training -p -w model-training'
      workingDirectory: 'model-training'

  - task: AcuraCI
    inputs:
      azureSubscription: 'ML'
      scriptLocation: 'inlineScript'
      inlineScript: 'az ml folder attach -n siliudelearn -p ML -w model-training'
      workingDirectory: 'model-training'

  - task: AcuraCI
    inputs:
      azureSubscription: 'ML'
      scriptLocation: 'inlineScript'
      inlineScript: 'az ml compute target delete -n ml-computetarget -w ml-computetarget'

Summary

The Azure Machine Learning workspace is an incredible tool for your AI projects. In a real-world AI project, you’re most likely work with multiple collaborators. You will have well-defined roles. Your data will need to be kept secure and you’ll have to worry about versions. That’s versions not just of your code but also your data, your experiments, details of all your deployments, created models, etc.

The Azure ML workspace automates all of this for you, and it records all of it behind the scenes for you as a part of your normal workflow. Later, if your customers come and ask you a question such as, “Hey why did you make such prediction at such a time,” you can easily trace your steps back to the specific deployment, specific algorithm, specific parameters, and specific input data that caused you to make that prediction.

Did you know that researchers once fooled a Google image recognition algorithm by replacing a single picture of a turtle, so Google would interpret it as a rifle? These kinds of attacks are new to AI. And the ML workspace helps you track all of this kind of thing very well. You still have to put in the work to secure your artifacts end to end, but the ML workspace is a great management tool.

Finally, I showed you how to automate this entire process end to end using an MLOps pipeline like you would do in any other software project. Until next time!
Create a WPF User Feedback Screen

Create a screen for the user to input feedback to your support department about your WPF application, as shown in Figure 1. On this screen, validate the data using the Entity Framework. The rules that fail in EF are going to be converted into validation messages to be displayed in the same manner as presented in the last article.

The user feedback screen (Figure 1) places the labels above each input field. The label styles in the StandardStyles.xaml file sets the margin property to 4. However, this would place the labels too far to the right above the input fields. You’re going to create a new style just on this screen to move the margin to the left. This style overrides the global Margin setting for labels. Open the UserFeedbackControl.xaml file and locate the <UserControl.Resources> element. Add a new key-value pair for labels.

```
<UserControl.Resources>
    <Style TargetType="Label">
        <Setter Property="Margin" Value="8,0,0,0"/>
    </Style>
</UserControl.Resources>
```

Remove the <StackPanel> with the text box and button in that you added in the previous article. There are two columns on this feedback screen; one for the large vertical "Feedback" column, and one for all the input fields. Add a <ScrollViewer> and a <Grid> within the <Border> as shown in the following code.

```
<ScrollViewer VerticalScrollBarVisibility="Auto">
    <Grid DataTemplate=
        {StaticResource #invDesign1/Grid1}>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <Grid RowDefinitions="1">  
          <Label Grid.Column="0" Margin="8,0,0,0">Comment</Label>
        </Grid>
        <Grid.RowDefinitions/>
    </Grid>
</ScrollViewer>
```

Add a Large Vertical Column

On the left side of this screen, there’s a raised area that you build using a <Border> with a linear gradient brush, a label, and an image. Build the large vertical column using the code shown in Listing 1. Add this code just below the closing <Border> element. Add a Grid for Input Fields

The second column (on the right of this screen) is where you place the area for the user to input data. Add a new <Grid> below the closing <Border> element. Add 10 row definitions for this new grid, as shown in Listing 2.

```
<Grid Grid.Row="1">
  <Border Background="#f4f4f4" CornerRadius="10">
    <Border.Background Stretch="Fill" />
    <StackPanel Background="White">
      <Label Content="Feedback" Style="{StaticResource Interna2}"
          Margin="10"/>
      <Image Source="/app/Feedback.png"/>
      <Button Content="Clear Message"
          Style="{StaticResource cmdButton}" IsDefault="True"/>
      <TextBox Content="*Contact Information*
          Text="Email Address"/
          Text="Phone Number"/>
      <TextBox Grid.Row="2" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="3" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="4" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="5" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="6" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="7" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="8" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="9" Text="Email Address"
          Text="Phone Number"/>
      <TextBox Grid.Row="10" Text="Email Address"
          Text="Phone Number"/>
    </StackPanel>
  </Border>
</Grid>
```

Try It Out

Run the application, log in as a valid user, and click on the Feedback menu item to display the screen. If you’ve done everything correctly, the screen should look like Figure 1.
Listing 4: Create the appropriate input entity class for the user feedback screen.

```java
using System; using System.Collections.Generic; using System.Collections.ObjectModel; using System.Windows.Input; namespace WPF_SampleDataLayer { public class UserFeedback { [Required] public string Name { get; set; } [Required] public string EmailAddress { get; set; } [Required] public string PhoneNumber { get; set; } [Required] public string Message { get; set; } } }
```

Listing 5: Convert EF validation objects to ValidationMessage objects

```csharp
public class ConvertValidationErrors { public Dictionary<string, List<string>> ConvertValidationErrors(string ErrorMessage) { var validationErrors = new Dictionary<string, List<string>>(); // Add the validation errors to the dictionary foreach (var error in ErrorMessage.Split(',')) { var trimmedError = error.Trim(); if (!string.IsNullOrEmpty(trimmedError)) { var errorParts = trimmedError.Split(':'); var propertyName = errorParts[0]; var errorProperty = errorParts[1]; if (!string.IsNullOrEmpty(propertyName) && !string.IsNullOrEmpty(errorProperty)) { validationErrors.TryAdd(propertyName, new List<string> { errorProperty }); } } } return validationErrors; } }
```

Modify the User Feedback View Model Class

Add a property named Entity to the UserFeedbackViewModel class to hold the data input on the screen. A Save() method is needed to submit the data to the database. If you are going to also add a stub of a SendFeedback() method in case you want to email the feedback to your support department.

Add the Entity Property

Open the UserFeedbackViewModel.cs file and add the following using statements at the top of this file.

```csharp
```

Add a property named EntityOfType to the UserFeedbackViewModel class, as shown in the code below.

```csharp
private UserFeedback _entity; public UserFeedback Entity { get { return _entity; } set { _entity = value; NotifyOfPropertyChange(() => Entity); } }
```

Update the SampleDcContext Class

For EF to select records from and modify data in the UserFeedback table, add a DbSet property in the SampleDcContext class. Open the SampleDcContext.cs file and add a new DbSet property.

```csharp
public virtual DbSet<UserFeedback> UserFeedbacks { get; set; }
```

Add a Method to Convert EF Validation Errors to ValidationMessage Objects

The Entity Framework uses data annotation attributes to generate validation errors automatically for you. It raises an error that contains a collection of validation errors. The structure of this collection doesn't lend itself well to data binding on a WPF screen, so we need to convert these validation errors to a collection of ValidationMessage objects.

Add a few using statements at the top of the SampleDcContext.cs file, as shown below.

```csharp
```

Add a method to the SampleDcContext class, Listing 5, to perform the conversion of the EF validation errors into a collection of ValidationMessage objects. This method takes the ErrorMessage and PropertyName properties from the Entity Framework object and assigns them to a new ValidationMessage object. This object is added to a list of ValidationMessage objects that is returned from this method.

Try It Out

Run the application and click on the Feedback menu item. Click the Send Feedback button without entering any data to ensure that the validation is working. Next, enter some text and try clicking on the Save button. You should see the feedback message on the screen.

Listing 6: The Save() method adds user feedback and reports validation errors

```csharp
public bool Save() { bool ret = true; try { // Add user feedback to database userFeedbacks.Add(_entity); } catch (Exception ex) { ret = false; } return ret; }
```

Listing 7: After saving, send a feedback message to your support department

```csharp
public bool SendFeedback() { bool ret = false; try { // Save/Validate the data if (Save()) { // Send the Feedback Message here } } catch (Exception ex) { ret = false; } return ret; }
```

Defining the User Feedback Data Layer

Once you have the new table created in the database, you need to perform three more steps to interact with this table.

Go back to the Server Explorer window and right mouse-click on the Table folder and select the Refresh menu to see the new table.

Add User Feedback to the Data Layer

You now have one of the new tables created in the database. You need to perform three more steps to interact with this table.

Add a Save Method

The data entered by the user on the User Feedback screen is going to be saved into the UserFeedback table you created. The Save() method, shown in Listing 6, uses the SampleDcContext class to attempt to add the data. If the data is correct, a new record is added to the table; if the data isn't correct, a DbEntityValidationException exception is thrown. Take the DbEntityValidationException object and pass it to the ConvertValidationMessages() method you wrote in the SampleDcContext class. Store the return result from this method into the ValidationMessages property on the view model. Set the IsValidVisibility property to true to display the validation messages.

Add a SendFeedback Method

You may want to send an email to a specific person when one of the feedback messages is stored in the UserFeedback table. The SendFeedback() method, Listing 7, is where you might perform this. This article isn't going to cover writing that code, but use the code shown in Listing 7 to save the data and display an informational message that the feedback message was sent.
valid information into each field and click the Send Feedback button again. Open the Sample database and check the UserFeedback table to see if the data was stored successfully.

A Design Pattern for Master/Detail Screens

The next screen you’re going to create is one to list, add, edit, and delete users, as shown in **Figure 2**. To accomplish this, build two separate user controls: one for the list of users, and one that displays the details for an individual user. These two controls will be placed onto the user maintenance control that you already built. As you build this screen, you’re going to create some generic classes to use for any CRUD screen that you need to create.

Listing 8: A ListView allows you to put buttons within any column you want

Listing 9: Load users into an ObservableCollection so any bound control gets notification of changes

Modify the User Maintenance View Model

Open the UserMaintenanceListViewModel.cs file and in the attributes of the `UserControl`, add a loaded event.

After the closing `</Grid.RowDefinitions>` element add the label and text box controls shown in **Listing 10**. In the view model you’re going to create in the next section, an Entity property is created for the type User. You can see that the type bindings on each text box control is bound to the Entity property followed by the name of a property in the User class.

After the labels and text box controls, add a stock panel (**Listing 11**) for the Undo and Save buttons. Use Image and TextBlock controls within each button control to present an image and text for the user for the Save and Undo functionality.

Create a User Detail View Model
In the UserMaintenanceDetailControl user control, you see that you’re binding to the properties of an Entity object. This Entity object is going to be in a view model for the

Getting the Sample Code
You can download the sample code for this article by following the instructions under the article and issue, or by visiting resources.devexpress.com. Then select “ADPWyDFDArticles” from the Category drop-down. Then select “A Design Pattern for Building WPF Business Applications - Part 2” from the Item drop-down.
Add a property named Entity that is of the type User. This is the Entity property that you bind to the controls on the UserMaintenanceDetail user control. Be sure to add a using statement for the WPF.Space.DataLayer namespace at the top of this file.

```csharp
private User_Entity = new User();
```

public User Entity

```csharp
get { return Entity; }
set
    Entity = value;
    RaisePropertyChanged("Entity");
```

Override the LoadUser method.

After loading the list of users, it would be nice to set the Entity property to the first item in the list. This causes the binding on the user detail control to display the values for the user in the bound text box controls. Override the LoadUsers method, call the base.LoadUsers method, and then check to ensure that the Users collection has some users. Set the Entity property to the first user in the Users collection. Setting this property causes the RaisePropertyChanged event to be fired. This, in turn, causes the UI to redisplay the new values on the detail screen.

```csharp
override void loadUsers()
{
    // Load all users
    base.LoadUsers();
    // Set initial user
    if (Users.Count > 0) { this.Entity = Users[0]; }
}
```

Modify the User Maintenance View Model.

Open the UserMaintenanceViewModel.cs file and change the Inheritance from UserMaintenanceListModel to UserMaintenanceDetailViewModel. You now have separate view models for each of the three user controls you’ve built. Because each view model inherits from the other, from the UserMaintenanceViewModel, you get all of the functionality from the detail and list view models.

```csharp
public class UserMaintenanceDetailViewModel : UserMaintenanceListViewModel
{
    ...
}
```

Try It Out

Run the application and click on the Users menu item. The user list and detail screen will now show and display the first user’s detail in the detail area. Click on other users in the list view control and you should see that the user detail area is updated with the new user information for each click you perform.

Create Toolbar

Just above the user list control, add a Toolbar control into which you add some buttons, as shown in Listing 12. The toolbar does not work yet, but you’ll add functionality to it in the next article.

Try It Out

Run the application and click on the Users menu item. Notice that the toolbar has been added.

Summary

In this article, you built the User Feedback Screen to allow a user to submit feedback about your application. While building this screen, you learned to work with the validation errors returned from the Entity Framework. You also learned to use control aggregation to build a screen from different user controls. Building an application this way allows you to test screen functionality separately. Using inheritance of the view models brings all the functionality for each of the user controls together so one user control can control all of the others. In the next article, you’ll learn to enable and disable each of the buttons based on what “state” you are in order to build the add, edit and delete functionality of the user screen.

Paul D. Sheffey

CODACode

SPONSORED SIDEBAR

Get .NET Core Help for Free!

Looking to create a new or convert an existing application to .NET Core or ASP.NET Core? Get started with a FREE Hour-long CodeConsulting session to make sure you get started on the right foot. Our consultants have been working with and contributing to the .NET Core and ASP.NET Core teams since the early pre-release builds. Leverage our team’s experience and proven track record to make sure that your next project is a success. For more information, visit www.codedmag.com/consulting or email us at info@codedmag.com.
Responsibly Packaged Management in Visual Studio

Almost nine years ago, a new open source project named NuGet (www.nuget.org) made its debut and two years after that debut, NuGet was and continues to be shipped with Microsoft Visual Studio. NuGet is one of several package managers, like Node Package Manager (NPM) for JavaScript and Maven for Java. Package Managers simplify and automate library consumption. For example, if you need a library to implement JavaScript Object Notation (JSON) capabilities in your .NET application, it takes a few clicks of the mouse and just like that, your application has powerful capabilities that you didn’t have to write, for free of charge.

Once upon a time, developers built and maintained their own libraries. If you needed a library, chances were, you asked fellow developers in online communities hosted in CompuServe in the giving spirit that was incident to such communities, and chances were good that you could get a library to meet your needs or, at the very least, you could get guidance on how to build it.

No production application or build process should ever take a direct dependency on any public package source.

Today, Open Source Software (OSS) has created an unprecedented availability to code and package management systems—so much so that code into your applications is a nearly frictionless process. That progress has ushered in not only numerous benefits, but also new risks and problems as well. One recent example: in November 2018 Event Stream incident involving NPM (https://bitly/2tgyA7I), the code in question was downloaded 20,000,000 times per week.

If you work for a public company governed by SOX or are subject to the Health Insurance Portability and Accountability Act (HIPAA) or Payment Card Industry (PCI) regulations, if your applications directly rely on a public NuGet source, there’s more than a fair chance that your company may be in violation of the aforementioned standards despite the lack of any adverse event.

In Case You’re Not Familiar with NuGet

If you’re not familiar with NuGet, what is it, and how it works, for additional context, you may want to consult the documentation at: https://docs.microsoft.com/en-us/nuget/what-is-nuget. If you want the comprehensive documentation PDF, you can download it here: https://bitly/nugredit70. If you’re a Visual Studio subscriber, you may want to watch my Introduction to NuGet Course: https://www.pluralsight.com/courses/nuget.

The concepts presented herein do not require an extensive NuGet understanding. The intended audience includes experienced developers as well as directors and managers tasked with implementing a company’s security and risk mitigation policies.

Package Managers and Package Sources

Before delving into the basic package manager concepts in .NET/Visual Studio with NuGet, let’s get some context on package managers and packages in general. The following are the core definitions you need to understand:

- **Package**: An archive file (i.e., a .zip or .tar file) that contains code artifacts and additional metadata used by a package manager that, in turn, is used by a development environment to add a package’s contents to a project.
- **Package Manager**: A tool that provides an application development environment (i.e., Visual Studio, Eclipse, etc.) used to gain access to packages contained in a package source. Common package managers are NuGet, Maven, and Node Package Manager (NPM). Not only does a package manager manage access to a specific package, it also manages the access to other packages that the downloaded package depends upon (dependency management).
- **Package Source**: A collection of packages that, for each package, contains metadata about that package. Such metadata includes the current version number, release history, links to the source code repository (i.e., GitHub), documentation, licensing information. Common package sources include NuGet.org, MyGet, and npml.org.

Companies should build and manage their own packages and the dependencies thereof, and create and use their own package source feeds.

The relationship among these three (NuGet.org, MyGet, and npml.org) is simple: Application development environments use package managers to connect to package sources and obtain packages to be used in an application development project.

What’s the Risk?

Of the three elements in the bullshit list above, risk arises from two: Package Managers and Package Sources. Package Sources like npml.org and NuGet.org are open environments to the extent that anybody can get an account and upload a package for others to download. For that reason alone, such open package sources are inherently untrustworthy. Does that mean you should avoid the use of such repositories? Of course not! Does that mean that when taking packages from such sources, you should perform the necessary due diligence to verify that package’s contents? Now that’s a no-brainer! The key question is, what’s the risk? The Event Stream incident discovered in November 2018, that included malicious code in a package that harvested account information from accounts having GitHub credentials of a certain level. The report (https://www.theregistrar.co.uk/2018/10/14/npm_repos_broken_channel/) that the code was downloaded at an average of 246,000 times per week.

If you can’t determine a package’s provenance and its contents with certainty, you’re exposing your firm to risk that could be otherwise mitigated.

On one hand, open package sources make code easily available. On the other, these open package sources are NOT certified. CAUTION: package sources for malicious content. Should who be policing packages? The answer is simple: YOU. If you bring a package into your organization, it’s your responsibility to verify not only the package’s contents, but the contents of every other package that the downloaded package depends upon.

Managing dependencies is another nice feature that a package manager provides. If you’re thinking that bringing a malicious package into your organization is like unleashing a virus, you’re getting the point.

The fact is, no production application or build process should ever take a direct dependency on any public package source. Setting aside malicious actors, there are many innocuous reasons to not trust public package sources:

- You’re leaving everything up to the package owner to manage versions and dependencies. What if the package owner introduces a dependency that makes the package work, but is completely incompatible with your application?
- What if the package owner uploads a new package version that works, but nevertheless introduces a bug into your application? If you set your build process to automatically upgrade your packages, you’re now introduced to what might be a costly bug that you’ll need to spend real money fixing.

Companies should build and manage their own packages and the dependencies therein and create and use their own package source feeds. If you leverage a package from a public source, you should open the package and evaluate its contents, and add that package to your own source feed or add the contents to your own package.

Doesn’t Package Signing Mitigate the Risk?

Yes, but it’s a qualified yes. Signing mitigates some risk, but not all risk. Signing wouldn’t have prevented the Event Stream Incident. The only thing package signing does is validate the package owner/contributor. Indeed, in most environments, you can limit which packages you can take to your own systems. If you’ve published the public key, then only those packages signed with the author’s certificate can be taken. However, that doesn’t mean you just take any package from that author. What if the author’s certificate was compromised? What if the author made an innocent mistake that ends up with your company sustaining some injury?

Now that we have a background on package managers, package sources, and the associated risks, let’s apply that knowledge to NuGet.

NuGet at a Glance: Creating Your Own NuGet Source

As previously stated, this article is not a comprehensive how-to on NuGet. To start, that is quite a dry technical subject. Like many software managers, and package sources in general, NuGet follows the same approach. In Visual Studio, there is the NuGet Package Manager, Illustrated in Figure 1.

If you leverage a Package from a public source, you should open the package, evaluate its contents, and add that package to your own source feed or add the contents to your own package.

Also illustrated in Figure 1 is the package source. Most likely, your active package source is NuGet.org. In my case, it’s something labeled Local Package Source. Figure 2 illustrates what that is:

As you can see, the Local NuGet Source is just a directory on my development computer. This may be news: Setting up a NuGet Source is as simple as creating a directory! Figure 3 illustrates the NuGet Packages in the directory.

The Anatomy of a NuGet Package

A NuGet Package is just a zip archive with a different extension (.nupkg). Figure 4 illustrates how the contents open.

Figure 5 illustrates the package contents. Let’s examine what are arguably the most popular and widely used NuGet Packages: Newtonsoft.Json, Microsoft.ContentModeling, and Microsoft.Azure.Cosmos.
Referring to Figure 5, the items of interest are the lib folder and the signature, license, and nuspec files:

- **lib folder**: This folder contains one or more subfolders that use a naming convention for each supported .NET version. You can learn more about targeting multiple .NET versions here: https://docs.microsoft.com/en-us/NuGet/create-package/supporting-multiple-target-frameworks.
- **signature.p7s file**: As the name implies, this is the signature file signed by the author's certificate. You can find more information on how to sign NuGet Packages here: https://docs.microsoft.com/en-us/NuGet/create-package/sign-package. You can learn more to require that only signed packages be accessible and to limit packages to certain authors here: https://docs.microsoft.com/en-us/NuGet/consume-packages/installing-signed-packages.
- **License.md**: This is a markdown file that contains the license terms and conditions for your package. Typically, this consists of an open source license such as the MIT, GNU, or Apache 2.0 licenses.
- **NuSpec file**: The NuSpec file is the manifest file. This is an XML file that is used to create the NuGet Package. This file will be discussed in the next section.

Creating Your Own NuGet Package

You now understand what Packages, Package Managers, and Package Sources are and have a basic understanding of how NuGet fits into that space. You also understand how to create and reference your own package source with nothing more than a directory of files. All that's left to get started is to learn how to create your own NuGet Package. To illustrate, I'm going to use the Immutable Class Library I created and wrote about a few issues back (https://www.codemag.com/Article/1905401/Immutability-in-C#).

There are several approaches you can use to create NuGet Packages. I'm going to show you the method I consider the
easiest to use and understand. There are also many options you can apply that I won’t cover here. For comprehensive coverage of all you can do with package creation, consult the documentation at NuGet.org.

**Step 1: Create a Package Directory Structure and Add Your Binaries**

Figure 6 illustrates the directory structure.

I added an Icon.png file that will be displayed in the Package Manager, as shown in Figure 1. The license text file contains the MIT License Language. Finally, there’s the nuspec file, which is illustrated in Figure 7.

**Step 2: Create Your Nuspec File**

Figure 7 illustrates the nuspec file as displayed in the NuGet Package Manager.

For a complete nuspec reference, you can find that information here: https://docs.microsoft.com/en-us/NuGet/reference/nuspec. The ID you choose for your package must be unique in the context of the source within which it’s hosted. Accordingly, if you elect to make your NuGet Package available in the NuGet.org feed, the ID must be unique in that universe. Figure 8 illustrates how the package appears in the NuGet Package Manager.

**Step 3: Create Your NuGet Package**

In order to create your NuGet Package from the command line, you need the NuGet Command Line Tools. Figure 9 illustrates where you can download NuGet.exe.

**Step 4: Publish Your Package**

Depending on the type of package source you are using, your steps may be slightly different. For a file directory source, the process is as simple as copying the file to the directory. If you’re hosting your own NuGet Server (https://docs.microsoft.com/en-us/NuGet/hosting-packages/NuGet-server), you will use one of the methods described here: https://docs.microsoft.com/en-us/NuGet/nuget-org/publish-a-package.

**Other Hosting Options**

Instead of self-hosting or using the NuGet.org public feed, you may instead elect to use a third-party service. For NuGet, there are paid services such as myget (myget.org) and Chocolatey (chocolatey.org). If it’s so easy to host your own feed, why would you consider a paid service? These paid services offer their own libG (Slurubar Recovery) infrastructure. If you host your own feed, you need to consider how your server will be backed-up and replicated and how you will recover in the event of a catastrophic event.

**Conclusion**

Open source has made it easier than ever to add features to your applications. Part of that ease is speed. Speed and ease mean less friction. Once upon a time, before open source as we know it today, before the Internet, and before package management, there was implicit friction in the system, which provided time to assess and evaluate. Developers of another generation, in my opinion, had a better understanding of change management. They understood the discipline and rigor required to mitigate risk. For all the benefits of today’s technology and the speed and ease we get with it, it’s more important than ever to employ risk mitigation techniques such as what is discussed in this article because if it’s easier for us to do good things, it’s easier for bad actors to use the same technology. Robot security and risk mitigation aren’t free. If there’s one negative side-effect of free open source, it’s the expectation that things happenfore with a cost no longer have a cost. Consider that the next time a package is introduced into your environment. If your organization is governed by SOX, HIPAA, FINRA, PCA, etc., if you’re compli-

Figure 10: The NuGet.exe pack command, using a nuspec file generates the NuGet Package.
Moving from jQuery to Vue

Most of the attention that JavaScript gets is all about creating large, monolithic Single Page Applications (SPAs). But the reality is that a great percentage of websites still use much simpler jQuery and vanilla JavaScript. Without going all-in on moving everything to a SPA, can you gain some of the benefits of using a framework to simplify your code and make it more readable and testable?

Sure, you can. In many cases, moving to a SPA framework means a complete re-thinking of your application. It's a change in how you approach building applications. I whole-heartedly recommend that you think about it this way if you're building new applications, as it can really change the way you approach Web development, but...

In many cases, it's beneficial to ramp up to these technologies. Tearing down your jQuery empire and adding something like Angular, Vue, or React is a big leap. That's one of the reasons I love how Vue works.

In this article, my goal is to give you, the jQuery user, a taste of the different approach that Vue takes and how this can improve your code, your markup, and your ability to build apps quickly. Yeah, quickly.

What's Wrong with jQuery?

Nothing's wrong with jQuery. Really, nothing. It's been pivotal to creating most of the great websites you know. jQuery was responsible for making cross-browser/cross-OS websites work. It's great.

But in many ways, it's getting long in the tooth. Let's take a quick example from Jake Rocheta’s blog post here: https://specbykey.com/blogging/simple-reddit-api-web-application-jquery/. He created a small example that uses jQuery to show how to call the GitHub API. You can see it in Figure 1.

The code is simple but shows a lot of the benefits and drawbacks of jQuery. Let’s break it down. You can see the JavaScript in Listing 1. Most of the code is inside one large event handler.

```javascript
$(document).ready(function() {
    ...}
```

Although this is pretty easy to remedy, it's a common practice because it's easy to think of an event handler as the main place for code in jQuery.

Next up is changing the UI in jQuery:

```javascript
$('#profile').html('<div id="loader"><div id="loading"></div><div id="loading"></div></div>');
```

It starts up immediately with code that builds up markup in code. This mixes the metaphors of UI and logic into a single file. Getting the markup correct with in-line text is notoriously fragile because there is no real syntax checking.

Next, the code using jQuery to read a value from an input:

```javascript
var username = $('input[name=login]').val();
```

Although this is straightforward, it requires you to use an ID on every input that you need to data out of. It also means a query over the entire DOM which can be slow, even at the fast speeds of jQuery. But I suspect if you're reading this article, you already know about the issues with jQuery. How can you make it better with Vue?

Using Vue in Place of jQuery

I think the best way to describe the way that Vue works is to convert this simple jQuery. You can see in Listing 1 the complete JavaScript of the project and Listing 2 contains the complete Markup. Let's rebuild this piece by piece so you can see how Vue can be used to do this easier and with less code.

Creating the Vue Object

One of the things I really like about Vue is that instead of requiring you buy into a big ecosystem, you can just drop a JavaScript file and start working with it. To get started, you'll just use a link to a development version of the library:

```html
<script src="https://cdn.jsdelivr.net/npm/vue@next"></script>
```

In fact, in this example, I left the jQuery in the project as I'll get back to using it a little later. Yeah, really.

The basis of how Vue works is a Vue object. I just created a new JavaScript file and started out with just an instance of the Vue object:

```javascript
var app = new Vue(
    el: '#app
```

This object takes a JavaScript object to specify options. The first part that's important is the data. This represents a selector for the parent element in the HTML that you're telling Vue to take over for. This is a key difference between jQuery and Vue. The magic of Vue is to be able to search through the DOM and find the elements you're interested in. Vue takes responsibility for a section of the DOM (or the entire DOM). So, in this case, you want set up the code to take over the element called:

```html
<div id="app">...
```
The key differentiator for Vue and jQuery is that instead of trying to modify the DOM, you’ll just keep your business logic in JavaScript and let the bindings in the HTML show the changes that happen to the data.

Executing the API

This simple example takes the name of the GitHub user and executes a network request. The jQuery version uses jQuery to execute the network requests. Vue purposely doesn’t attempt to do the networking for you. You’re free to use any networking library you want. It could be jQuery (because you likely already know it), but there are also alternatives like axios and even a Vue-specific one called vue-resource. But because you’re converting code, let’s just keep the jQuery networking code.

The existing code uses jQuery’s getJSON to get the response from the API. Instead of constructing HTML like the old version does (see Figure 1), you’re going to just create a new data property called user that will hold the object from the server.

Inside the value for v-on, you can simply define what code to execute. In this example, you can just call a method on the Vue object. So now you can add the onSubmit method on the Vue object like so:

```javascript
<var app v-text="text">
  "Hello, world!"
</var>
```

### Collection Binding

In jQuery, you’re used to creating collections of markup in order to create lists and grids. For example, from the jQuery version:

```javascript
<% var repositories = $jsonRepositories(index); %>
<% var repos = 0 %>
<% for (var i = 0; i < repositories.length; i++) { %>
  <div class="git Repositories" target="_blank">
    <% repos++; %>
    <% repository = repositories[i]; %>
    <a href="${repository(index).url}"
      class="btn bg-info info-info在现场" target="_blank">
      <% repository = repos %>
      <% repos = 1 %>
    </div>
  </a>
</div>
```

The v-for attribute instructs the element to create an anchor tag for every repository in the user.repos property and display it with the `href` attribute set to the repository’s URL.
Quality Software Consulting for Over 20 Years

CODE Consulting engages the world’s leading experts to successfully complete your software goals. We are big enough to handle large projects, yet small enough for every project to be important. Consulting services include mentoring, solving technical challenges, and writing turn-key software systems, based on your needs. Utilizing proven processes and full transparency, we can work with your development team or autonomously to complete any software project.

Contact us today for your free 1-hour consultation.

Helping Companies Build Better Software Since 1993

www.codemag.com/consulting
832-717-4445 ext. 9 • info@codemag.com

Moving from jQuery to Vue

to create a local variable called repo to represent the individual repository.

When using v-for, you also need to specify a unique key for each record (which helps Vue remove items from the list when you change the collection in the code). You do that by binding to a key attribute with v-bind. In this case, you know that the URL is unique, so just use it as a key. You could use a primary key or other unique key. It doesn’t matter as long as it’s unique.

The rest of the code is just simple binding like you’ve seen before (setting the html and the contents of the anchor).

You can see the complete finished code in Listings 3 and 4.

Where Are We?

As a jQuery user, you may be used to being able to create simple applications by dropping jQuery and writing some quick code. I would argue that Vue is a better choice these days. It allows you to quickly get your job done and keep the code from becoming a spaghetti mess of callbacks and inline HTML. But it does require a learning curve that can be a little steep at first. The switch of mindset from querying and manipulating the DOM to having the DOM react to changes in your code is a key change. But once you get over that hump, I think you’ll see the benefit of using a framework like Vue.
Why Use GraphQL?
GraphQL was developed to make reusing the same API into a flexible and efficient process. GraphQL works for API clients with varying requirements without the server needing to change implementation as new clients get added or removed, and the client needing to change how they use the API when new features are added to it. It solves many of the inefficiencies that you may experience when working with a REST API.

Some of the reasons you should use GraphQL when building APIs are:

- GraphQL APIs have a strongly typed schema
- No more over- or under-fetching
- Analytics on API use and affected data

Let’s take a look.

A Strongly Typed Schema as a Contract Between the Server and Client
The GraphQL schema, which can be written using the GraphQL Schema Definition Language (SDL), clearly defines what operations can be performed by the API and the types of data that will be returned. This is key in the server’s validation engine that validates requests from clients to determine if the operation can be executed.

No More Over- or Under-Fetching
GraphQL has a declarative way of requesting data using the GraphQL query language syntax. This way, the client can request any data it wants, as long as those types and their fields are defined in the schema. This is analogous to RESTful APIs where the endpoints return predefined and fixed data structures.

This declarative way of requesting data solves two commonly encountered problems in RESTful APIs:

- Over-fetching
- Under-fetching

Over-fetching happens when a client calls an endpoint to request data, and the API returns the data the client needs as well as extra fields that are irrelevant to the client. An example to consider is an endpoint `/users/id` which returns a user’s data. It returns basic information, such as (in this example, an online school’s database will be used) name and department, as well as extra information, such as address, billing information, or other pertinent information such as courses they’re enrolled in, purchasing history, etc. For some clients or specific pages, this extra information can be irrelevant. A client may only need the name and some identifying information, like social security number or the courses they’ve enrolled in, making the extra data such as address and billing information irrelevant. This is where over-fetching happens, affecting performance. It can also consume more of users' Internet data plans.

Under-fetching happens when an API call doesn’t return enough data, forcing the client to make additional calls to the server to retrieve the information it needs. If the API endpoint `/users/id` only returns data that includes the user’s name and one other bit of identifying data, clients needing all of the user’s information (billing details, address, courses completed, purchasing history, etc.) will have to request a lot of data, which should be loaded in separate API calls. This affects performance for these types of clients, especially if they’re on a slow connection.

This problem isn’t encountered in GraphQL applications because the client can request exactly the bits of data they need from the server. If the client requirement changes, the server need not change its implementation but rather the client is updated to reflect the new data requirement by adding the extra fields() it needs when querying the server. You will learn more about this and the declarative query language in GraphQL in the upcoming sections.

Analytics on clients’ usage
GraphQL uses resolver functions (which I’ll talk about later) to determine the data that the fields and types in the schema return. Because clients can choose which fields the server should return with the response, it’s possible to track how those fields are used and evolve the API to decrease fields that are no longer requested by clients.

Setting Up the Project
You’ll be building a basic GraphQL API that returns data from an in-memory collection. Although GraphQL is independent of the transport layer, you want this API to be accessed over HTTP so you’ll create an ASP.NET Core project.

Create a new ASP.NET Core project and install the dependencies shown in Figure 1.

The first package you installed is the GraphQL package for .NET. It provides classes that allow you to define a GraphQL service and a GraphQL engine to execute GraphQL queries. The second package provides an ASP.NET Core middleware that exposes the GraphQL API over HTTP. The third package is referred to as the GraphQL Playback, which works in a similar way to Postman for REST APIs. It gives you an editor in the browser where you can write GraphQL queries against your server and see how it responds. It gives you an Intellisense and you can view the GraphQL schema from it.

The GraphQL Schema
The GraphQL schema is at the center of every GraphQL server. It defines the server’s API, allowing clients to know which operations can be performed by the server. The schema is written using the GraphQL schema language (also called schema definition language, SDL). With it, you can define object types and fields to represent data that can be retrieved from the API as well as root types that define the group of operations that the API allows. The root types are the Query type, Mutation type, and Subscription type, which are the three types of operations that you can run on a GraphQL server. The query type is compulsory for any GraphQL schema, and the other two are optional. Although you can define custom types in the schema, the GraphQL specification also defines a set of built-in scalar types. They are Int, Float, Boolean, String, and ID.

There are two ways of building GraphQL server applications. There’s the schema-first approach where the GraphQL schema is designed up front. The other approach is the code-first approach where the GraphQL is constructed programmatically. The code-first approach is common when building a GraphQL server using a typed language like C#. You’re going to use the code-first approach here and later look at the generated schema.

Let’s get started with the schema. Create a new folder called GraphQL and add a new file Book.cs with the content in the following snippet:

```csharp
public class Book
{  
    int Id { get; set; }  
    string Title { get; set; }  
    string Pages { get; set; }  
    string Chapters { get; set; }  
}
```

Add another class BookType.cs and paste the content from the next snippet into it.
The code in the last snippet represents a GraphQL object type in the schema. It will have fields that will match the properties in the Book class. You set the Pages and Chapters fields to be nullable in the schema. If not set, by default, the GraphQL .NET library sets them as non-nullable.

The application you’re building only allows querying for all the books and querying for a book based on its ID. The book type is defined so go ahead and define the root query type. Add a new file RootQuery.cs in the GraphQL folder, then copy and paste the code from Listing 1 into it.

The RootQuery class will be used to generate the root operation query type in the schema. It has two fields, book and books. The book field returns a list of Book objects, and the book field returns a Book type based on the ID passed as an argument to the book query. The type for this argument is defined using the IDGraphType, which translates to the built-in ID scalar type in GraphQL. Every field in a GraphQL type can have zero or more arguments.

You will also notice that you’re passing in a function to the Resolve parameter when declaring the fields. This function is called a Resolver function, and every field in GraphQL has a corresponding Resolver function used to determine the data for that field. Remember that I mentioned that GraphQL has an execution algorithm! The implementation of this execution algorithm is what transforms the query from the client into actual results by moving through every field in the schema and executing their Resolver function to determine its result.

The books resolver calls the GetBooks() static function to return a list of Book objects. You’ll notice that it’s returning a list of Book objects and not BookType, which is the type tied to the schema. GraphQL for .NET library takes care of this conversion for you.

The Book resolver calls context.GetArgument with id as the name of the argument to retrieve. This argument is then used to filter the list of books and return a matching record.

The last step needed to finish the schema is to create a class that represents the schema and defines the operation allowed by the API. Add a new file GraphSchema.cs with the content in the following snippet:

```csharp
using GraphQL;
using GraphQL.Types;

public class GraphSchema : Schema
{
    public GraphSchema(IDependencyResolver resolver) : base(resolver)
    {
        Query = resolver.ResolveRootQuery();
    }
}
```

In that bit of code, you created the schema that has the Query property mapped to the RootQuery defined in Listing 1. It uses dependency injection to resolve this type. The IDependencyResolver is an abstraction over whichever dependency injection container you use, which, in this case, is the one provided by ASP.NET.

Let’s move on to the Configure method to add code that sets up the GraphQL server and also the GraphQL middleware that is used to test the GraphQL API. Add the code snippet below to the Configure method in Startup.cs.

```csharp
app.UseGraphQL<GraphSchema>.
    UseGraphQL<GraphSchema, Options>.
```

The GraphQL Query Language

So far, you’ve defined the GraphQL schema and resolvers and have also set up the GraphQL middleware for ASP.NET.
which runs the GraphQL server. You now need to start the server and test the API.

Start your ASP.NET application by pressing F5 or running the command dotnet run. This opens your browser with the URL pointing to your application. Edit the URL and add /api/playground at the end of the URL in order to open the GraphQL playground, as you see in Figure 2.

The GraphQL playground allows you to test the server operations. If you’ve built REST APIs, think of it as a Postman alternative for GraphQL.

Now let’s ask the server to list all the books it has. You do this using the GraphQL query language, another concept of GraphQL that makes it easy for different devices to query for data how they want, served from the same GraphQL API.

Go to the GraphQL playground. Copy and run the query you see on the left side of the editor in Figure 3, then click the play button to send the query. The result should match what you see on the right-hand side of the playground, as shown in Figure 3.

You’ll notice that the query is structured similarly to the schema language. The books field is one of the root fields defined in the query type. Inside the curly braces, you have the selection set on the books field. Because this field returns a list of Book type, you specify the fields of the Book type that you want to retrieve. You omitted the pages field; therefore it isn’t returned by the query.

You can test the book(id) query to retrieve a book by its ID. Look at Figure 4 and run the query you see there to retrieve a book. In that query, you set the id argument to a value of 3, and it returned exactly what you need. You’ll notice that I have two queries, books and book(id: 3). This is a valid query. The GraphQL engine knows how to handle it.

What's Next?
So far, I’ve covered some basics of GraphQL. You looked at defining a schema using the code-first approach, writing resolver functions, and querying the GraphQL API. You created a server using the GraphQL package for .NET, the NuGet package GraphQL.Server.Transports.AspNetCore, which is the ASP.NET Core middleware for the GraphQL API, and GraphQL.Server.Ui.Playground package that contains the GraphQL playground. You used the GraphQL playground to test your API. I explained that in GraphQL, there are three operation types. In this article, you worked with the query operation; in the next article, you’ll look at mutations and accessing a database to store and retrieve data. You’ll update your schema so you can query for related data, e.g., authors with their books, or books from a particular publisher. Stay tuned!!

Peter Meisner

The CODE brand is widely-recognized for our ability to use modern technologies to help companies build better software. CODE is comprised of five divisions - CODE Consulting, CODE Staffing, CODE Framework, CODE Training, and CODE Magazine. With expert developers, a repeatable process, and a solid infrastructure, we will exceed your expectations. But don’t just take our word for it - ask around the community and check our references. We know you’ll be impressed.

Contact us for your free 1-hour consultation.

Helping Companies Build Better Software Since 1993
Design Patterns for Distributed Systems

Containers and container orchestrators have fundamentally changed the way we look at distributed systems. In the past, developers had to build these systems nearly from scratch, resulting in each architecture being unique and not repeatable. We now have infrastructure and interface elements for designing and deploying services and applications on distributed systems using reusable patterns for microservice architectures and containerized components.

Today’s world of always-on applications and APIs has availability and reliability requirements that would have been required of only a handful of mission-critical services around the globe only a few decades ago. Likewise, the potential for rapid, viral growth of a service means that every application has to be built to scale nearly instantly, in response to user demand. These constraints and requirements mean that almost every application that’s built, whether it’s a consumer mobile app or a back-end payments application, would benefit from a distributed system. But building distributed systems is challenging.

Containerized building blocks are the basis for the development of reusable components and patterns that dramatically simplify and make accessible the practices of building reliable distributed systems. Reliability, scalability, and separation of concerns dictate that real-world systems are built out of many different components spread across multiple computers. In contrast to single-node patterns, multi-node distributed patterns are more loosely coupled throughout the patterns dictate communication between the components, this communication is based on network calls. Furthermore, many calls are issued in parallel, and systems coordinate via lose synchronization rather than tight constraints.

Microservice Architecture

Recently, the term microservices has become a buzzword for describing multi-node distributed software architectures. Microservices describe a system built out of many different components running in parallel via networked service bindings. APIs. Microservices stand in contrast to monolithic systems like that in Figure 1, which tend to place all of the functionality for a service within a single, tightly coordinated application.

Figure 2. Instead, illustrates how individual functions can be separated on isolated services and interact with each other through a programming interface (API).

There are numerous benefits to the microservices approach, most of which are centered around reliability and agility. Microservices break down an application into small pieces, each focused on providing a single service. This reduced scope enables each service to be maintained as a single agile team. Reduced team size also reduces the overhead associated with keeping a team focused and moving in one direction.

Additionally, the introduction of formal APIs in between different microservices decouples the teams from one another and provides a reliable contract among the different services. This formal contract reduces the need for tight synchronization among the teams because the team providing the API understands the surface area that it needs to keep stable, and the team consuming the API can rely on a stable service without worrying about its details. This decoupling enables teams to independently manage their coding and release schedules, which, in turn, improves each team’s ability to iterate and improve their function.

Finally, the decoupling of microservices enables better scaling. Because each component has been broken out into its own service, it can be scaled independently. It’s rare for each service within a larger application to grow at the same rate; often, some services scale up while others scale down. Without microservices, stateless and scale can simply scale horizontally, whereas other systems maintain state and require sharding or other approaches to scale. By using microservices, you can use the approach to scaling that suits it best. This isn’t possible when all services are part of a single monolith.

Azure Kubernetes Service

Kubernetes (https://kubernetes.io) is a rapidly evolving platform that manages container-based applications and their associated network and storage dependencies. The focus is on the application workload and not the underlying infrastructure components. Kubernetes provides a declarative approach to deployments, backed by a robust set of APIs for management operations.

You can build and run modern, portable, microservices-based applications that benefit from Kubernetes orchestrating and managing the availability of those application components. As an open platform, Kubernetes allows you to build your applications with your preferred programming language, OS, libraries, or messaging bus. Existing continuous integration and continuous delivery (CI/CD) tools can integrate with Kubernetes to schedule and deploy releases.

Azure Kubernetes Service (AKS): https://azure.microsoft.com/en-us/services/kubernetes-service/) provides a managed Kubernetes service that reduces the complexity for deployment and core management tasks, including coordinating updates. The AKS cluster masters are managed by the Azure platform, and you only pay for the AKS pods that run your applications.

Figure 3 shows some basic components of a Kubernetes cluster architecture that’s useful to get familiar with, because describing some typical design patterns for distributed systems and how to implement them in AKS. First of all, a Kubernetes cluster is divided into two components:

- Master cluster nodes that provide the core Kubernetes services and orchestration of application workloads.
- Nodes that run your application workloads.

To run your applications and supporting services, you need a Kubernetes node. An AKS cluster has one or more nodes, which is an Azure virtual machine that runs the Kubernetes node components and container runtime. The Kublet is the Kubernetes agent that processes the orchestration requests from the cluster master and manages the node components. The container runtime is the component that allows containerized applications to run and interact with additional resources, such as the virtual network and storage.

Replicated Load-Balanced Services

Replicated load-balanced services is probably one of the simplest distributed patterns, and one that most are familiar with when considering a replicated load-balanced application. In such an application, illustrated in Figure 4, every service is identical to every other service and all are capable of supporting traffic. The pattern consists of a scalable number of services running in a load balancer in front of them. The load balancer is typically either completely round-robin or uses some form of session stickiness.

Stateless services are ones that don’t require a saved state to operate correctly. In simplest stateless applications, even individual requests may be routed to separate instances of the service. Stateless systems are replicated to provide redundancy and scale.

To create a replicated service in Azure Kubernetes Service, you can spin up the virtual node to provision and connect to your workloads. With virtual nodes, you have fast provisioning of pods, and only pay per second for their execution time. In a scaling scenario, you don’t need to wait for the Kubernetes cluster autoscaler to deploy VM compute nodes to run the additional pods. A Kubernetes pod is a group of containers that are deployed together on the same host. If you have defined a single container, you can generally replace the word “pod” with “container” and accurately understand the concept.

If the resource needs of your application change, you can manually scale an AKS cluster to run a different number of nodes. When you scale down, nodes are carefully cleansed and drained to minimize disruption to running applications. When you scale up, AKS waits until nodes are marked ready by the Kubernetes cluster before pods are scheduled on them.

To scale the cluster nodes, first get the name of your node pool using the azaks show command. The following command gets the node pool name for the cluster named my-Cluster in the mfsResourcesGroup resource group:

```
azaks show --resource-group mfsResourcesGroup
-mycf clusters --name my-Cluster --output json
```

This is a way to manually scale an AKS cluster to increase or decrease the number of nodes. You can also use the cluster autoscaler (currently in preview in AKS) to automatically scale your cluster. The autoscaler can watch for pods in your cluster that can’t be scheduled due to resource constraints. When issues are detected, the number of nodes is increased to meet the application demand. The autoscaler can also be configured to scale the number of nodes based on the CPU, memory, or disk consumption of the cluster. The autoscaler can be configured to scale the number of nodes based on the CPU, memory, or disk consumption of the cluster.
Microservice Architecture

Microservices describe a system built out of different components running in different processes and communicating via defined APIs.

Azure Kubernetes Services

AKS provides a managed Kubernetes service that reduces the complexity for deployment and management.

To use the cluster autoscaler, you need the `aks-preview` extension version 0.4.1 or higher. Install the `aks-preview` extension using the `az extension add` command, then check for any available updates using the `az extension update command`.

Sharded Services

Replicating stateless services improves reliability, redundancy, and scaling. Within a replicated service, each replica is entirely homogeneous and capable of serving every request. Another design pattern emerges in contrast to replicated services, called shared services. Figure 6 illustrates how, with shared services, each replica, or shard, is only capable of serving a subset of all requests. A load-balancing module is responsible for examining each request and distributing each request to the appropriate shard for processing.

Replicated services are generally used for building stateless services, whereas shared services are generally used for building stateful services. The primary reason for sharing the data is because the size of the state is too large to be served by a single computer. Sharding enables you to scale a service in response to the size of the state that needs to be served.

Applications that run in Azure Kubernetes Service may need to store and retrieve data. For some application workloads, this data storage can use local fast storage on the node that is no longer needed when the pod is deleted. Other application workloads may require storage that persists in more regular data volumes within the Azure platform. Multiple pods may need to share the same data volumes or reattach data volumes if the pod is rescheduled on a different node. Finally, you may need to inject sensitive data or application configuration information into pods.

A good practice to reduce management overhead and let your code is not manually create and assign persistent volumes, as this would add management overhead and limit your ability to scale. The recommendation, instead, is to use dynamic storage provisioning by defining the appropriate policy to minimize unneeded storage costs once the pods are deleted, and allow your applications to grow and scale as needed. Figure 7 refers to a persistent volume claim (PVC) that lets you dynamically create storage as needed.

To define different tiers of storage, such as Premium and Standard, you can create a Storage Class. The Storage Class also defines the reclaim policy. This reclaim policy controls the behavior of the underlying Azure storage resource when the pod’s deletion is requested. The persistent volume should be retained when the pod is deleted. This behavior is controlled by the `managed-premium-retain` flag which can be specified in the YAML file containing the configuration parameters.

A persistent volume represents a piece of storage that has been provisioned for use with Kubernetes pods. A persistent volume can be used by one or many pods and can be dynamically or statically provisioned. As before, a storage class is used to define how a unit of storage is dynamically created with a persistent volume. For more information on Kubernetes storage classes, see Kubernetes Storage Classes (https://kubernetes.io/docs/concepts/storage/storage-classes/). The following command shows the pre-created storage classes available within an AKS cluster:

```
$ kubectl get sc
```

Modern application development often aims for stateless applications, but stateful applications need to be used for stateful applications, such as applications that include database components. A StatefulSet is similar to a deployment in that one or more identical pods are created and managed. Replicas in a StatefulSet follow a graceful, sequential approach to deployment, scale, upgrades, and terminations. For more information, see Kubernetes Stateful Sets (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/).

Scatter-Gather Pattern

So far, we’ve examined systems that replicate for scalability in terms of the number of records processed per second (the stateless replicated pattern), as well as scalability for the size of the data (the stateless pattern with the shard or the data). The scatter-gather pattern in Figure 8 uses replication for scalability in terms of time and allows you to achieve parallelism in servicing requests, enabling a significant increase in throughput as long as the service node can handle the load.

Like replicated and shared systems, the scatter-gather pattern is a tree pattern with a root that distributes requests and leaves that process those requests. However, in contrast to replicated and shared systems, scatter-gather systems are simultaneously fanned out to all of the replicas in the system, and the persistence data set is a small amount of processing and the root server then returns a fraction of the result to the root. The root server then combines the various partial results together to form a single complete response to the request and then sends this request back out to the client.

Scatter-gather is quite useful when you have a large amount of mostly independent processing that’s needed to handle a...
Nest.js Step-by-Step: Part 2

In the first part of this series (https://www.codemag.com/Article/1907081/Nest.js-Step-by-Step), you were introduced to Nest Framework and you started building the To Do REST API using mock data and mock data for development and testing isn't sufficient to make a realistic and ready-to-launch app. Using a database to store your data is part of the process and is mandatory for making a great launch. In this article, I’ll use an instance of PostgreSQL database running locally on a Docker container. To access the database and execute the queries and mutations, I’ll make use of TypeORM, which is one of the most mature Object Relational Mapping (ORM) tools in the world. Nest.js comes with built-in support for TypeORM.

The source code for this article series is available on this GitHub repository: https://github.com/bhalidar/nestjs-todo-app/ and online at CODE Magazine, associated with this article.

I’ll start by introducing TypeORM and its features, then explore how Nest.js integrates with TypeORM. Finally, the step by step demonstration shows how to convert the code from Part 1 into code that is database-aware and eliminates the use of any mock data.

Nest.js can deal with a rich variety of databases ranging from relational databases to NoSQL ones.

What is TypeORM?

TypeORM is a JavaScript library that’s capable of connecting to several database engines, including PostgreSQL, Microsoft SQL Server, and MongoDB, just to name a few. By hiding the complexity and specificity of connecting to different database engines, TypeORM enables the communication between your application and the back-end database of your choice.

TypeORM is built on top of TypeScript decorators that allow you to decorate your entities and their corresponding properties so that they map to a database table with columns.

TypeORM supports both the Active Record and Data Mapper patterns. I won’t be touching on these topics, but you can read more about them by following this link: https://medium.com/oozax/geeks/the-active-record-and-data-mappers-of-orm-pattern-estb62676bc.

In general, I prefer using the Data Mapper pattern and specifically using Repositories to access the database. TypeORM supports the repository design pattern, so each entity has its own Repository object. These repositories can be obtained from the database connection itself.

In addition, TypeORM allows you to create your own custom repository by letting you extend the standard base repository and add any custom functions that you need.

Here’s a quick summary of TypeORM features that I’m going to use in the application you’re about to build:

- Entity Decorator to mark a JavaScript as an entity in the database
- Column Decorator to customize the mapping between a JavaScript object property and the corresponding column in the database. Customization includes specifying column data type, length, allow null or not, and other useful settings.
- A repository object per entity by auto-generating them. You can inject those objects in your Nest.js services and start accessing the database.
- Table relationships including one-to-one, one-to-many, and many-to-many relationships.

In this article, I’ll be using mostly one-to-many and many-to-many relationships.

I strongly advise you give it a try if you’re serious about using TypeORM in your professional projects. Access everything about TypeORM at https://typeorm.io.

How Nest Framework Integrates with TypeORM?

The (nestjs)/TypeORM package is a Nest.js module that wraps around the TypeORM library and adds a few services providers into the Nest.js Dependency Injection system. The following is a list of services that are added by default:

- TypeORM database Connection object
- TypeORM Entity manager object (used with data mapper pattern)
- TypeORM Repository object per entity (for each entity defined in the application)

Every time a service or controller in your application injects any of the above services, Nest.js serves them from within its Dependency Injection system.

You can check the source code for this module by following this URL: https://github.com/nestjs/TypeORM.

Demo

“The proof of the pudding is in the eating!”

This is a famous saying that has always haunted me since I started software development. To best understand the concepts that I’m going to explain, I recommend that you grab the source code from Part One of this series (in CODE Magazine July/August 2019), and follow along to apply it as you go. Another saying is practice makes perfect, so let’s start.

Setup Docker & PostgreSQL database

Step 1: Open the application in your favorite editor and create a new git branch to start adding database support.

git checkout -b add-db-support

Step 2: Install the NPM packages that are required to run the application and connect to the database by issuing the following command:

```yarn add @nestjs/typeorm pg```

This command installs three NPM packages:

- @nestjs/typeorm is the Nest.js module wrapper around TypeORM.
- TypeORM is the official NPM package for TypeORM library.
- pg is the official library connector for PostgreSQL database.

Step 3: Setup a docker-compose file to run an instance of PostgreSQL on top of Docker. Docker is a prerequisite you have to have on your computer in order to run this step.

```yarn add -D services```

The docker-compose file you'll use for the application is simple. The file defines a single service called db. The Docker file defines the following settings for the db service:

```services:
  db:
    container_name: todo_db
    image: postgres:10.7
    volumes:
      - db/initdb:/docker-entrypoint-initdb.d
    ports:
      - 5432:5432```

To run any initialization script when Docker first creates the container, you can place the script file inside the /db/initdb directory and Docker automatically runs it upon creating the container.

By defining a Docker volume mapped to a local directory, you have more control to your Docker container

Setting up a new bash script file inside the /db/initdb directory. Listing 1 shows the content of this file.

The script starts by including a shawbang line that makes it an executable script. The executable is straightforward. It starts by running a new database user todo and a new database todo. It also makes the todo user a SuperUser and assigns it the password todo123.

```$ docker run -it -u $USER postgres:latest
CREATE USER todo WITH ENCRYPTED PASSWORD 'todo123';
GRANT ALL PRIVILEGES ON DATABASE todo TO todo;```

If you can’t find the docker-utils utility on your computer, you can download it from this URL: https://sourceforge.net/projects/docker-utils/.

Step 6: Create the Docker container and build your database. Let’s start by adding a new NPM script to the package.json file to make the task of starting up Docker and creating the container easier to add. Add the following script:

```yarn run start:services```

This command creates the Postgres container, a new PostgreSQL database, and a database user.

Step 7: Verify that the database is up and running by opening a new command line session and running the following docker command:

docker exec -it todo_db bash```

The command above starts an interactive docker session on Postgres container instance that you’ve just created. By default, running this command opens a bash session for you to run and execute your commands. To verify that the database exists, run the following commands:

- Connect to Postgres engine: pg-"-l postgres
- List the existing database: "1"

You should be able to see an entry for the todo database.

Listing 3: Bash File

```#!/bin/bash
cd /app
codemag.com```

Next.js Step-by-Step Part 2
Configure TypeORM

Step 1: Initialize TypeORM by creating a new ormconfig.json file at the root of your application. Simply paste the JSON content in Listing 2 into the JSON file:

TypeORM expects this configuration file to create a connection to the database, deal with database migrations, and all things related. TypeORM supports a variety of means to store the connection options including JSON, JS, XML, and YAML files, and also environment variables.

You can read more about the TypeORM connection options by following this URL: https://github.com/typeorm/TypeORM/blob/master/docs/using-ormconfig.md.

The most important connection options that you need for the application are:

- **Name**: The name of the configuration settings. You can have one named "development" and another named "production". You need one set of configurations per running environment.

- **Type**: The type of the database that TypeORM is connecting to. In this case, the type is "postgres".
- **Host, port, username, password, and database**: These settings resemble the details of the connection string that TypeORM uses to connect to the underlying database.
- **Synchronize**: A value of true means that TypeORM will auto-synchronize to the application code and the database structure every time the application runs. This is good for development but you should avoid using it in production. Preferably, you should create database migrations for each and every change you make in the application code, even when in development. Make it a habit, it's guaranteed a smooth transition when you are ready to deploy your app and database to production.
- **Logging**: If you enable this setting, TypeORM will emit some logging messages on the application's console when it's running. This is helpful in the development phase.
- **Entities**: This is the path where TypeORM finds the entities your application is using and maps to a table in the database.
- **Migrations**: This is the path where TypeORM finds the migrations you create and runs them against the database.

Step 2: Amend the tsconfig.json file of your application to look like the content of Listing 3.

The next few steps will make heavy use of the @todo path, as you will see.

With Nest.js, there's a variety of ORMs to use in your application. You can even develop your own integration with any other model and make it work inside Nest.js.

Step 3: Change all references to TodoEntity and TaskEntity in the application code to match the ones here:

```typescript
import { TodoEntity } from './todo.entity';
import { TaskEntity } from './task.entity';
```

Later, when you start creating migrations, TypeORM requires having the same exact reference path to all entities stored in the database. For instance, if the application code references the same entity in two different places with two different paths, TypeORM assumes that these are two different entities. To make sure that TypeORM is happy, use the same path for all entities all over the application code.

Step 4: I'm making use of a few helper methods that wrap TypeORM API calls. Create a new src/shared/utils.js file. Copy the functions from Listing 4.

```typescript
const { getConnectionOptions, getConnection } = require('typeorm');
export const getConnectionsOptions = async (connectionName) => {
  let options = await getConnectionOptions({
    process.env.NODE_ENV || 'development',
  })
  return options
}
```

The getDbConnectionOptions() function reads the TypeORM configuration settings from the ormconfig.json file based on the current Node environment. Remember back in Step 8, when you assigned a name for the configuration settings object as development? This is a convenient trick that proves helpful, especially when you move the application to production. So you can easily add another configuration object with the name production.

The getDbConnection() function retrieves a connection from TypeORM.

The runMigrations() function runs the pending migrations using the active database connection.

Step 5: Let's change the AppModule and make it return a DynamicModule by accepting TypeORM connection settings.

Locate the src/app.module.ts file and replace its content with the code in Listing 5.

The module now defines a forRoot() method that accepts the connection options and returns a DynamicModule.

The DynamicModule is a normal Nest.js module that you return and customize the way you want. You can read more about DynamicModule on: https://docs.nestjs.com/modules.

Step 6: Import the TypeORMModule by replacing the content of AppModule with the code in Listing 6.

You simply import the TypeORMModule into the imports section of the AppModule and passing the connection options as follows:

```typescript
import { TypeORMModule, forRoot, createConnection, MappedColumn } from 'typeorm';
```

```typescript
import { TodoEntity } from './todo.entity';
import { TaskEntity } from './task.entity';
```

```typescript
import { getConnectionsOptions } from 'typeorm';
```

The code makes use of the help function defined to load the TypeORM connection options based on the current executing environment and then it feeds the results to the AppModule.forRoot() method.

Step 6: Let's automate the process to run database migrations while the application is bootstrapping.

By default, Nest.js expects you to generate or create your own database migrations using TypeORM CLI commands and then run them against the database.

Open the src/main.ts file and add the following line of code just beneath the code block that creates the AppModule:

```typescript
const app = await NestFactory.create(AppModule);
```

With the code below:

```typescript
** 'Run DB migrations'
const runMigrations = async () => {
  await runMigrations();
}
```

```typescript
runMigrations();
```
TypeORM

TypeORM is an ORM that can run in Node.js, Browser, Cordova, PhoneGap, Ionic, React Native, Netcore, Expo, and Electron platforms and can be used with TypeScript and JavaScript (ES5, ES6, ES7, and ES8).

You defined the runDBMigrations() method back in Step 4 of the Configure TypeORM section. Now when the application is bootstrapping and before it starts listening to new HTTP requests, it runs any pending migration against the database and makes sure that the application entity model is in sync with the database table model.

This is exactly what happens when you set synchronize: "true" in the ormconfig.json file. However, being able to decide when to run migrations puts you in the driver’s seat with greater control on migrations.

Define Entity Relationships

Step 1: Let’s convert the TodoEntity and TaskEntity objects to real TypeORM entities.

The rule is simple. To map a JavaScript entity object to a PostgreSQL database table, you decorate the entity object with @Entity(name: string) decorator offered by TypeORM APS. You can read up on TypeORM entities here: https://github.com/TypeORM>TypeORM/blob/master/docs/entity.md.

To map a property on the entity object to a column on the database table, you decorate the property with @Column() decorator, also offered by TypeORM APS. Listing 7 shows both entities decorated and ready to be used by TypeORM to create their corresponding database tables. There are many decorators offered by TypeORM.

For example, you have relation decorators to define a relation between one entity and another. Listener decorators to react to events triggered by TypeORM and many others. Here’s a list of all decorators supported by TypeORM: https://github.com/TypeORM>TypeORM/blob/master/docs/decorator-reference.md.

TodoEntity defines a property tasks of type array of TaskEntity. Here, the relation is that one TodoEntity has one or many TaskEntities. Hence the code uses @OneToMany() decorator for this purpose.

On the other side of the relation, TaskEntity defines a property todo of type TodoEntity. One Task belongs to one and only one TodoEntity.

```javascript
Listing 7: TodoEntity and TaskEntity classes
import { TodoEntity } from './todoEntity/task.entity';

import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, OneToMany } from 'typeorm';

@Entity('todo')
export class Todo {
  id: number;
  name: string;
  description: string;
  createdAt: Date;
  updatedAt: Date;

  @ManyToOne(type => TodoEntity, todo => todo.tasks, { cascade: true })
  todoEntity: TodoEntity;
}

import { TaskEntity } from './todoEntity/task.entity';

@Entity('task')
export class Task {
  id: number;
  name: string;
  description: string;
  createdAt: Date;
  updatedAt: Date;

  @ManyToOne(type => TodoEntity, todo => todo.tasks, { cascade: true })
  todoEntity: TodoEntity;
}
```

Now, to generate a migration, let’s first add an NPM script so that you don’t have to write a long command every time you want to generate a new migration.

```javascript
Listing 8: TodoController class
app();

app.get('/new', (req, res) => {
  res.render('todo/new');
});

app.post('/create', (req, res) => {
  const todo = new Todo();
  todo.name = req.body.name;
  todo.description = req.body.description;
  todo.create();
  res.redirect('/');
});
```

```javascript
Listing 9: DTO classes
export class TodoDto {
  id: number;
  name: string;
  @CreateDateColumn()
  createdAt: Date;
  @UpdateDateColumn()
  updatedAt: Date;
}
```

The command compiles the entity objects in the application to the corresponding database tables (if already present) and then generates the necessary steps so that both models are in sync.

Step 4: Run the migrations!

 Previously, you’ve configured the application to run any pending migrations during the bootstrapping phase.

To run the new migrations that you’ve generated, simply start the application and migrations will run automatically.

To run the application, issue the following command:

```javascript
Listing 10: migration:generate InitMigration
```

Step 5: Change the TodoController class. There’s no change required except decorating the update() and create() methods with the @UsePipes() decorator.

In NestJS, Pipes allow you to transform data from one format to another. You can even use Pipes to perform data validation on the input that the client passes with the HTTP Request.

Next, come with two built-in Pipes: ValidationPipe and ParseUrlPipe. You use the former to add validation over the input parameters. Use the latter to validate and convert an input parameter into a valid integer value.

You can also create your own custom Pipe. See the article on pipes for more:

For this project, you’re going to use ValidationPipe. But first, you need to install the Pipe by passing a validation decorator that the class-validator library offers. Visit: https://github.com/oclus/class-validator.

Listing 9 shows all the DTO objects annotated with the proper validation rules.

You can check the validation annotations on TaskDto and TaskCreateDto objects on the GitHub repository of this article or by checking out the online version of this article.

Use Repositories within your Services

Amend the TodoService class. There are major changes to this class to adjust the way the code manages data. Instead of using an in-memory array to store the To Do items and tasks, you’ll vary that to connect directly to the database.

You start by injecting the Repository<TodoEntity> instance as follows:

```javascript
Listing 11: CreateTodo
class TodoController {
private readonly todoPipe: Repository<TodoDto> = createTodoPipe();

@Post(
'/create',
{ consumeInBody: true, transformInBody: transformInBody() },
{ usePipes: [createTodoPipe()] }
)
async createTodo(@Body() todo: TodoDto) {
  return this.todoService.createTodo(todo);
}
}
```

The @nestjs>TypeORM defines the @InjectRepository() decorator. Its role is to retrieve a Repository instance for a specific entity from the NestJS Dependency Injection system and make it available for the service.
Listing 10: getOneTodo() function

```typescript
async getOneTodo(todoId: string): Promise<TodoEntity> {
  const todo = await this.todoRepo.findOne(
    where: { id: todoId },
    relations: ["tasks", "owner"],
  );
  if (!todo) throw new HttpException();
  return todo;
}
```

Listing 11: toTodoDto() function

```typescript
@OneToMany(() => TaskEntity, task => task.todo, {
  cascade:
    true,
})
```

Listing 12: createTodo() function

```typescript
async createTodo(todoDto: TodoCreateDto): Promise<TodoEntity> {
  const { name, description } = todoDto;
  const todo: TodoEntity = await this.todoRepo.create(
    name,
    description,
  );
  await this.todoRepo.save(todo);
  return toTodoEntity(todo); // convert to TodoEntity
}
```

Earlier, when you imported TypeORMModule.forRoot(["TodoEntity, TaskEntity") into the module, Nest.js registered a Repository service provider (token and factory method) for each and every Entity in the Nest.JS Dependency Injection System.

Next, you’ll look at a few methods. The rest you can find in the source code accompanying this article at the GitHub repository and in the online version of this issue of CODE Magazine.

Let’s take a look at the toTodoDto() utility function that maps a TodoEntity to a TodoDto object. You could use some other advanced libraries like AutoMapper but for this article, I’ve decided to keep it simple and create my own mapper function. Listing 11 shows the toTodoDto() function.

Listing 12 shows the code for the TodoService createTodo() function.

The repository object exposes the create() function to create a new instance of an entity. Once you create a new instance of TodoEntity, you save the entity in the database by using another function exposed by the repository object, which is the save() function.

Conclusion

In this article, you’ve seen how easy it is to connect a Nest.js application to a database using the TypeORM library.

In the upcoming article, you’ll be looking at adding and dealing with users and authentication modules.

Happy Nesting!

Bilal Haidar

A best practice is to always inject Repositories inside your Services rather than working directly with Repositories inside your Controllers.

Listing 10 below shows how to implement getOneTodo() method. The method uses the findOne() function, available on the Repository instance, to query for a single TodoEntity based on the TodoId. In addition, it returns all of the related TaskEntity lists on this object. If the entity isn’t found in the database, the code throws an HttpException. Otherwise, it converts the TodoEntity object into a TodoDto object and returns the data to the calling controller.
Cross-Platform Mobile Development Using Flutter

For a number of years, mobile developers have had to grapple with maintaining multiple code bases of their apps—one for each platform. And for a number of years, that meant developing simultaneously for iOS, Android, Windows Phone, and even BlackBerry. Fortunately, that didn’t last. Today, the mobile platform wars yielded two winners: iOS and Android.

Even so, developers dread having to maintain dual code bases for their apps unless it’s totally necessary. Companies are also trying to avoid maintaining multiple code bases; otherwise they need to have separate teams of developers specializing in each platform.

In recent years, cross-platform development frameworks have emerged as the life savers for developers, with Xamarin taking the lead with its Xamarin suite of development frameworks for cross-platform mobile development. And more recently, Facebook’s React Native proves to be a hit with mobile developers, allowing developers to create mobile apps using JavaScript, a language that’s already familiar to a lot of full-stack developers.

Not wanting to be left out of the burgeoning mobile market, in late 2018, Google announced Flutter 1.0, its latest cross-platform framework for developing iOS and Android apps. In this article, I’ll give you an introduction to Flutter. By the end of this article, you’ll be on your way to developing some exciting mobile apps using Flutter!

Getting Started with Flutter

Flutter is Google’s portable UI toolkit for building natively compiled mobile, Web, and desktop apps using Google’s Dart programming language.

Flutter has the following major components:

- **Flutter engine**: Written in C++, provides low-level rendering support using Google’s Skia graphics library
- **Foundation Library**: Written in Dart, provides a large layer of functionality for apps and APIs to communicate with the engine
- **Widgets**: Basic building blocks for UI

In the next couple of sections, I’ll show you how to install Flutter and start writing your first Flutter application. Once you’ve gotten started with the basics, you’ll create a news reader application that demonstrates how easy it is to write compelling mobile apps with Flutter.

Installing Flutter

To develop cross-platform iOS and Android mobile apps with Flutter, you need to use a Mac. For this article, I’m going to base my examples on the Mac. Before you get started, you need to ensure that you have the following components installed:

- Xcode
- Android Studio

To install Flutter on your Mac, head over to this page: https://flutter.dev/docs/get-started/install/macos. The instructions on this page are pretty clear and self-explanatory, and I won’t repeat them here.

For the development environment, you can use Android Studio or Visual Studio Code. I prefer Visual Studio Code. To configure Visual Studio Code to support your Flutter development, check out this page: https://flutter.dev/docs/development/tools.vs-code.

Creating Your First Flutter Project

Once the SDK and tools are set up, you are ready to create your first Flutter application. The easiest way is to type the following command in Terminal:

```bash
$ flutter create hello_world
```

Note that Flutter project names must be in lower case and you can use the underscore character (_) (if you need to use a separator for the project name (just don’t use camel case). The above command creates a folder named `hello_world` containing a number of files foreseeing your project.

To examine the content of the Flutter project created for you, open the `hello_world` project using Visual Studio Code.

You can open up your Flutter project by dragging the project folder into Visual Studio Code.

![Image of Visual Studio Code showing the files in the Flutter project](image)

Figure 1 shows the content of the Flutter project.

Of particular interest are the following files/folders:

- **main.dart**: The main file in the lib folder. This is the main file of your Flutter application.
- **ios**: This folder contains the iOS application that runs on your iOS device/simulator.
- **android**: This folder contains the Android application that runs on your Android device/emulator.
- **pubspec.yaml**: This file contains references to the various packages needed by your application.

To run the application, you need the following:

```yaml
import 'package:flutter/material.dart';
```

Android SDK built for 8+ - emulator:
- [android-sdk](https://developer.android.com/studio) 8.0.0
- **iOS**: Xcode 11.2
- [Java](https://www.oracle.com/java/technologies/javase-jdk11-downloads.html) 11
- [Node.js](https://nodejs.org/en/) 12.16.0
- **Android**: Android Studio 3.5.1
- [emulator](https://developer.android.com/studio) 8.0.0

**To run the application on a particular device, use the following command:**

```bash
$ flutter run --device id 04524480f50
```

The `--device_id` is highlighted.

When the application has successfully loaded onto the simulator or emulator, you should see them, as shown in Figure 2.

Understanding How Flutter Works

To learn how Flutter works, it’s good to look at the `main.dart` file in the `hello_world` project and see how the various components work. Frankly, it’s not the easiest way to learn Flutter because the various statements in the file can be quite overwhelming for the beginning developer. That’s why I’ll start off with the bare minimum and build up the application from scratch.

**Widgets**

Unlike other cross-platform development frameworks (like Xamarin and React Native), Flutter doesn’t use the platform’s native widgets. For example, in React Native,
Flutter doesn't rely on the device's OEM widgets. It renders every view's components using its own high-performance rendering engine.

Types of Widgets
In Flutter, there are two main types of widgets:

- **Stateless widgets**: Changing the properties of stateless widgets has no effect on the rendering of the widget.
- **Stateful widgets**: Changing the properties of stateful widgets triggers the life cycle hooks and updates the UI using the new state.

Before you look at how to create stateless and stateful widgets, let's erase the entire content of the `main.dart` file and replace it with the following statements:

```dart
import 'package:flutter/material.dart';

void main() => runApp(
  MaterialApp(
    title: 'Hello, CODE Mag!',
    body: Center(
      child: Text('Hello, CODE Mag!'),
    ),
  ),
);
```

Hot-reload has no effect on the root widget; in general, when you perform a hot-reload, the `main()` function won't be re-executed and no changes will be observed.

In this example, `Container` (which is a widget) is the root widget of the application. As the name implies, the `Container` widget is used to contain other widgets, and in this case, it contains the `Center` widget, which, in turn, contains the `Text` widget and displays the string “Hello, CODE Mag!”

If you’ve run the application previously from Terminal, you don’t need to stop the application in order for the application to be updated. Flutter supports two types of updates:

- **Hot reload (press ‘r’ in Terminal)**: This option allows you to update the UI without restarting the application.
- **Hot restart (press ‘R’ in Terminal)**: This option allows you to restart the application.

Figure 4 shows what happens when you press “R” to hot-restart the application. For this example, hot-reload has no effect as all of the UIs are defined in the root widget. You’ll see hot-reload in action later on when I discuss stateless and stateful widgets.

Figure 5 shows the application running on the simulator and emulator.

Using the `MaterialApp` and `CupertinoApp` Classes
The example in the previous section has a dark background and doesn’t look like a traditional iOS or Android application. Flutter provides two main convenience widgets that wrap your widgets in the design styles for the iOS and Android platforms:

- **MaterialApp**: The `MaterialApp` class represents an application that uses material design. It implements the Material design language for iOS, Android, and Web.
- **CupertinoApp**: The `CupertinoApp` class represents an application that uses Cupertino design. It implements the current iOS design language based on Apple’s Human Interface Guidelines.

Let’s now wrap the widget using the `MaterialApp` class:

```dart
import 'package:flutter/material.dart';

void main() => runApp(
  MaterialApp(
    title: 'Material App Demo',
    home: Scaffold(
      appBar: AppBar(
        title: Text('Material App Demo'),
      ),
      body: Center(
        child: Container(
          margin: const EdgeInsets.all(10.0),
          color: Color(0xFFF00000),
          width: 100.0,
          height: 100.0,
          child: Text('Hello, CODE Mag!'),
        ),
      ),
    ),
  ),
);
```

Figure 4: Performing a hot restart in Terminal

Figure 5: The application running on the iOS Simulator and Android emulator
Listing 1: Creating a stateless widget

```dart
void main() => runApp(CupertinoApp(),
  title: 'Cupertino App Demo',
  home: CupertinoPageScaffold()
    navigationBar: CupertinoNavigationBar(),
    middle: const Text('Cupertino App Demo'),
  );

class MyCustomWidget extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text('Hello, World!'),
    );
  }
}
```

Listing 1 shows the previous UI rewritten as a stateless widget.

Stateless Widgets

So far, you have a pretty good idea of how UI in Flutter is created using widgets. In the previous section, the UI was created all in the `runApp()` function. A much better way to build the UI is to “compartilheize” the widget into independent widgets so that they can be reused. Now let’s try to reorganize the code so that the UI is written as a stateless widget.

To create a stateless widget:

- Name the new Widget class and extend it from `StatelessWidget`.
- Implement the `build()` method, with one argument of type `BuildContext` and return type `Widget`.

Here is the template for a stateless widget:

```dart
class MyCustomWidget extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text('Hello, World!'),
    );
  }
}
```

Hot restart the application and you should see the same output as shown in Figure 7.

Now, add another instance of the `MyCustomWidget` to the `main.dart` file:

```dart
void main() => runApp(CupertinoApp(),
  title: 'Cupertino App Demo',
  home: CupertinoPageScaffold()
    navigationBar: CupertinoNavigationBar(),
    middle: const Text('Cupertino App Demo'),
  );

class MyCustomWidget extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text('Hello, World!'),
    );
  }
}
```

Do you still remember about the hot-reload that I mentioned earlier? Modify the color in the stateless widget as follows:

```dart
Listing 1 shows the previous UI rewritten as a stateless widget.

Stateless Widgets

So far, you have a pretty good idea of how UI in Flutter is created using widgets. In the previous section, the UI was created all in the `runApp()` function. A much better way to build the UI is to “compartilheize” the widget into independent widgets so that they can be reused. Now let’s try to reorganize the code so that the UI is written as a stateless widget.

To create a stateless widget:

- Name the new Widget class and extend it from `StatelessWidget`.
- Implement the `build()` method, with one argument of type `BuildContext` and return type `Widget`.

Here is the template for a stateless widget:

```dart
class MyCustomWidget extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text('Hello, World!'),
    );
  }
}
```

Hot restart the application and you should see that there are now two instances of `MyCustomWidget` (see Figure 8).

Figure 1: Applying the `MaterialApp` class to the application

Figure 2: Applying the `CupertinoApp` class to the application

Figure 3: Applying the `CupertinoPageScaffold` class to the application

Figure 4: Applying the `CupertinoNavigationBar` class to the application

Figure 5: Applying the `CupertinoPageScaffold` class to the application

Figure 6: Applying the `MaterialApp` class to the application

Figure 7: Applying the `CupertinoApp` class to the application

Figure 8: Displaying two instances of `MyCustomWidget`
When you now hot reload the app (press "r" in Terminal), you’ll see the colors of the MyCustomWidget change immediately (see Figure 9).

### Stateful Widgets
Stateless widgets are useful for displaying UI elements that don’t change during runtime. However, if you need to dynamically change the UI during runtime, you need to create stateful widgets.

Stateful widgets don’t exist by themselves. They require an extra class to store the state of the widget. To create a stateful widget:

1. Name the new Widget class and extend it from StatefulWidget.
2. Create another class that extends from the State class, of the type that extends from the StatefulWidget base class. This class will implement the build() method, with one argument of type BuildContext and return type of Widget. This class will maintain the state for the UI to be updated dynamically.
3. Override the createState() function in the StatefulWidget subclass and return an instance of the State subclass (created in the previous step).

The following shows the template for creating a stateful widget:

```dart
class MyCustomStatefulWidget extends StatefulWidget {
  // constructor with named
}
```

Using the earlier example, let’s now create a stateful widget by appending the code in bold (as shown in Listing 2) to main.dart.

To make use of the stateful widget, add it to the runApp() function, like this:

```dart
import 'package:flutter/cupertino.dart';

void main() => runApp(
');
```

### Building the app

CupertinoApp(
  title: 'Cupertino App Demo',
  home: CupertinoPagesStatefulWidget()
)

Using the following code in bold (as shown in Listing 3) to main.dart.

The following shows the widget tree in the stateful widget:

```dart
<widget>.
```

Performing a hot restart yields the UI, as shown in Figure 10. Clicking on the blue strip increments the counter. 

Observe the following:

- The MyCustomStatefulWidget class has a property named country. This value is initialized through the named argument in the constructor: MyCustomStatefulWidget((Key key, this.country)).
- The country property is used in the _DisplayState class, and it can be referenced by prefixing it with the widget keyword.
- Our stateful widget tree contains the widgets, as shown in Figure 11.
Building the News Reader Project

By now, you should have a good understanding of how Flutter works. The best way to learn a new framework is to build a simple app and see how the various components fall in place, so let’s now build a complete working application.

For this project, you’ll create a news application that displays the news headline in a ListView, as shown in Figure 12.

When the user taps on a particular headline, the application navigates to another page and loads the details of the news in a WebView (see Figure 13).

For fetching the news headlines, you can use the following API: https://newsapi.org/v2/top-headlines?country=us&category=business&apiKey=api_key.

You can apply your own free News API key from https://newsapi.org.

Creating the Project

Let’s start by creating the project:

```dart
// flutter create news_reader
```

Adding the Package

For this project, you need to use the HTTP package so that you can connect to the News API. Add the following statement in bold to the pubspec.yaml file:

```yaml

dependencies:
  flutter:
    sdk: flutter

Once you save the changes to the pubspec.yaml file, Visual Studio automatically fetches the package and installs it in your local drive. Alternatively, you can use the following command to manually download the packages:

```bash
flutter packages get
```

Importing the Packages

In the main.dart file, add the following statements in bold:

```dart
import 'package:flutter/material.dart';
// for future class
import 'dart:async';
```

Accessing the News API

The News API returns a JSON string containing a summary of the various news headlines. The first thing you need to do is to examine the structure of the JSON result returned and see which of the parts you need to retrieve for your application.

Accessing the News API using https://newsapi.org/v2/top-headlines?country=us&category=business&apiKey=api_key

Once the result is obtained, paste the result into a JSON formatter, such as http://jsonformatter.github.io. Figure 14 shows the JSON result formatted.

In particular, you’re interested in extracting the following:

• All of the articles referenced by the articles key
• For each article, extract the values of title, description, url, and urlToImage

Populating the ListView

Add the statements in bold to the main.dart file as shown in Listing 3.

A Future object represents the results of asynchronous operations

Here is what you’ve added to the main.dart file:

• You created a variable named _news and initialized it as a map object with the one key, articles, and set it to an empty list. Later you’ll connect it to the News Reader project.
You can now test the application on the iOS Simulator and Android emulator. Type the following command in Terminal:

```sh
cd news_reader
flutter run -d all
```

The applications should now look like Figure 15.

### Implementing Pull-to-Refresh
The next thing to do is to implement pull-to-refresh so that you can update the news feed by pulling down the ListTile and then releasing it. Add the statements in bold to the main.dart file as shown in Listing 4.

To ensure that the ListTile supports pull-to-refresh, use the RefreshIndicator widget and set its child to the ListTile.

```dart
import 'package:flutter/material.dart';
// for future class
import 'dart:async';
// for http
import 'package:http/http.dart' as http;
// for JSON parsing
import 'dart:convert';

// To store the data to pass to another widget
class NewsContent {
  final String url;
  NewsContent(this.url);
}

void main() {
  runApp(MyApp());
}
```

The NewsContent class is used to store the URL of the article so that it can be passed to the details page. Append the following block of code to the end of the main.dart file:

```dart
class DetailsPage extends StatelessWidget {
  final NewsContent data;

  DetailsPage({this.data})
    ..title = 'Details Page';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Container()
    );
  }
}
```

### Customizing the Content of the ListTile
Instead of displaying an image and the name of the news article on each row, it would be better to display the news title and its description, followed by a smaller image. Add the statements in bold to the main.dart file as shown in Listing 5.

### Converting to a Navigational Application
Now that the ListTile displays the list of articles, it would be nice if the user could tap on an article to read more about it. For this, you’re going to create a details page that will be used to display the content of the article.

Add the following statements in bold to the main.dart file:

```dart
class NewsContent {
  final String url;
  NewsContent(this.url);
}
```

Add the NewsContent class to the URL of the article so that it can be passed to the DetailsPage widget. Append the following block of code to the end of the main.dart file:

```dart
class DetailsPage extends StatelessWidget {
  final NewsContent data;

  DetailsPage({this.data})
    ..title = 'Details Page';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Container()
    );
  }
}
```

You can now test the application on the iOS Simulator and Android emulator. Type the following command in Terminal:

```sh
cd news_reader
flutter run -d all
```

The applications should now look like Figure 15.

### Implementing Pull-to-Refresh
The next thing to do is to implement pull-to-refresh so that you can update the news feed by pulling down the ListTile and then releasing it. Add the statements in bold to the main.dart file as shown in Listing 4.

To ensure that the ListTile supports pull-to-refresh, use the RefreshIndicator widget and set its child to the ListTile.

```dart
import 'package:flutter/material.dart';
// for future class
import 'dart:async';
// for http
import 'package:http/http.dart' as http;
// for JSON parsing
import 'dart:convert';

// To store the data to pass to another widget
class NewsContent {
  final String url;
  NewsContent(this.url);
}

void main() {
  runApp(MyApp());
}
```

The NewsContent class is used to store the URL of the article so that it can be passed to the details page. Append the following block of code to the end of the main.dart file:

```dart
class DetailsPage extends StatelessWidget {
  final NewsContent data;

  DetailsPage({this.data})
    ..title = 'Details Page';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Container()
    );
  }
}
```

You can now test the application on the iOS Simulator and Android emulator. Type the following command in Terminal:

```sh
cd news_reader
flutter run -d all
```

The applications should now look like Figure 15.

### Implementing Pull-to-Refresh
The next thing to do is to implement pull-to-refresh so that you can update the news feed by pulling down the ListTile and then releasing it. Add the statements in bold to the main.dart file as shown in Listing 4.

To ensure that the ListTile supports pull-to-refresh, use the RefreshIndicator widget and set its child to the ListTile.

```dart
import 'package:flutter/material.dart';
// for future class
import 'dart:async';
// for http
import 'package:http/http.dart' as http;
// for JSON parsing
import 'dart:convert';

// To store the data to pass to another widget
class NewsContent {
  final String url;
  NewsContent(this.url);
}

void main() {
  runApp(MyApp());
}
```

The NewsContent class is used to store the URL of the article so that it can be passed to the details page. Append the following block of code to the end of the main.dart file:

```dart
class DetailsPage extends StatelessWidget {
  final NewsContent data;

  DetailsPage({this.data})
    ..title = 'Details Page';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Container()
    );
  }
}
```

You can now test the application on the iOS Simulator and Android emulator. Type the following command in Terminal:

```sh
cd news_reader
flutter run -d all
```

The applications should now look like Figure 15.

### Implementing Pull-to-Refresh
The next thing to do is to implement pull-to-refresh so that you can update the news feed by pulling down the ListTile and then releasing it. Add the statements in bold to the main.dart file as shown in Listing 4.

To ensure that the ListTile supports pull-to-refresh, use the RefreshIndicator widget and set its child to the ListTile.

```dart
import 'package:flutter/material.dart';
// for future class
import 'dart:async';
// for http
import 'package:http/http.dart' as http;
// for JSON parsing
import 'dart:convert';

// To store the data to pass to another widget
class NewsContent {
  final String url;
  NewsContent(this.url);
}

void main() {
  runApp(MyApp());
}
```
Listing 6: Creating the Details Page

```dart
Listing 6

```}

---

**CODE Framework: Business Applications Made Easy**

Architected by Markus Egger and the experts at CODE Magazine, CODE Framework is the world's most productive, maintainable, and reusable business application development framework for today's developers. CODE Framework supports existing application interoperability or can be used as a foundation for ground-up development. Best of all, it's free, open source, and professionally supported.

Download CODE Framework at [www.codemag.com/framework](http://www.codemag.com/framework)

Helping Companies Build Better Software Since 1993

www.codemag.com/framework
832-717-4445 ext. 9 • info@codemag.com

---

**Application Details Page**

The DetailsPage takes the data passed into it (which is the URL of the article) and displays it in the center of the page.

Add the statements in bold to the main.dart file that are shown in Listing 6.

The onTap argument specifies that when the user taps on a row in the ListView, it navigates (using the Navigator.push() function) to the next page (DetailsPage) and passes it the data (NewsContent).

Redeploy the application and select a particular news headline and you should see the details page as shown in Figure 17.

**Adding a WebView**

Displaying the URL of the article in the details page is not very useful to the reader. What you really want to do is use a WebView to display the content of the article.

Add the following bolded statement to the pubspec.yaml file:

```yaml
dependencies:
  flutter:
    sdk: flutter
  http:
    path: http
  webview_flutter:

  The above statement adds the webview_flutter package to the project.

To use WebView on iOS, you need to add the following bolded statements to the Info.plist file located in the ios/Runner folder:

```xml
<key>UIWebViewConfiguration</key>
<dict>
  <!-- Set UIWebViewsettings for iOS. Please refer to the,
  // documentation of UIWebViewConfiguration for further
  // details. -->
  <key>allowsInlineMediaPlayback</key>
  <true/>
  <key>additionalMediaTypes</key>
  <array>
    <string>video/mp4</string>
  </array>
  <key>crossOriginPolicy</key>
  <string>use-credentials</string>
</dict>

```
Add the following bolded statements to the main.dart file:

```dart
import 'package:flutter/material.dart';

// for Future class
import 'dart:async';

// for http
import 'package:http/http.dart' as http;

// for JSON parsing
import 'dart:convert';

import 'package:webview_flutter/webview_flutter.dart';
import 'package:flutter_spinkit/flutter_spinkit.dart';

...}

class DetailsPage extends StatefulWidget {
  final NewContent data;

  DetailsPage({Key key, @required this.data}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Details Page'),
      ),
      body: WebView(child: DonutModel(url: data.html, javascripthMode: JavascriptMode.unrestricted),),
    );

    return MultiWidget();
  }
}

Summary
Learning a new framework is always challenging. But I do hope that this article has made it easier for you to get started with Flutter. Let me know what you are using now (Xcode, Android Studio, Xamarin, or React Native) and if you plan to switch over to Flutter. You can reach me on Twitter (@weimenglee) or email me at weimenglee@learn2develop.net.

Listing 7: Rewriting DetailsPage as a stateful widget

```dart
import 'package:flutter/material.dart';

// for Future class
import 'dart:async';

// for http
import 'package:http/http.dart' as http;

// for JSON parsing
import 'dart:convert';

import 'package:webview_flutter/webview_flutter.dart';
import 'package:flutter_spinkit/flutter_spinkit.dart';

...}

class DetailsPage extends StatefulWidget {
  final NewContent data;

  DetailsPage({Key key, @required this.data}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Details Page'),
      ),
      body: WebView(child: DonutModel(url: data.html, javascripthMode: JavascriptMode.unrestricted),),
    );

    return MultiWidget();
  }
}
```
Add File Storage to Azure App Services: The Work-Around

Azure App Service Plans in which Azure App Services run include 1, 10, 50, or 250 GB of disk space, depending on the pricing tier you choose, but if you need more than that, or if you don’t want to pay the premium price just for extra disk space, you’re out of luck. I recently worked on a project upgrading a website that required hundreds of GB of files to be stored in a virtual directory in the file system. When retrieved, some of the files referred to other files via a relative path. In addition, the files had to be protected by authentication and authorization. We had to maintain the hierarchy of the file system and integrate security so they could only be retrieved through the Web application, so we couldn’t just store the files in Blob storage.

You can’t add drive space or additional drives to Azure Web Services.

We soon found out that the obvious solution, to create a large file share in Azure and attach it to the App Service Plan, isn’t possible. You can’t add drive space or additional drives to Azure App Services. We spent many hours searching for a solution and found that this is, in fact, a highly requested feature in Azure and that quite a lot of us would like to see a solution. At one point, we came across a new Azure feature, now in Preview, to do just what we needed, but soon found out that it’s only going to be available in Linux-based App Services. So close! Undaunted, we came up with a work-around.

The Work-Around

Our application is written in ASP.NET MVC hosted on IIS on Azure APP Services, but the principles can be applied to many Web platforms. To re-state the problem, the browser sometimes requests files from the server via a path to a virtual directory. Many of the files returned from the server contain relative paths to additional files and the files can’t be modified, so the relative paths must continue to work. The amount of disk space available to the App Services is far less than the amount we require. The goal is to make the server respond to the unmodified URLs as though it’s retrieving files from a virtual directory (a very large one).

The first hurdle we faced was preventing the Web server from attempting to find and retrieve the files because they won’t be on the local drive where the website resides. The fundamental purpose of Web servers, after all, is to retrieve requested HTML and other files and return them to the browser. What we needed to happen instead was for IIS to allow me to handle the request. As far as IIS (the underlying technology used to host App Services in Azure) goes, when it encounters file extensions at the end of a URL, it assumes that you want to retrieve a file (through a process called request filtering) and goes to the file system looking for it. This happens very early in the processing pipeline, long before the request gets to ASP.NET, let alone any code under my control. Luckily, IIS has a way to alter this behavior for a specific path. Because we had to modify how IIS works, the change had to be made in the web.config file, as shown in Figure 1.

The configuration change shows how to override IIS’s default request filtering. In our case, all of the files in question resided in a virtual directory named Media. With this configuration change, any URL path that consists of the domain, followed by /Media/, followed by anything, using any of the HTTP verbs (something we only need the GET verb) will be passed to the TransferRequestHandler, which is the default handler for ASP.NET MVC. That means that when the browser asks for anything in this path, instead of looking for the file on disk and returning it, the server turns the request over to ASP.NET MVC, which looks at the route to determine what to do next.

In order to alleviate the need for an action parameter in the URLs for retrieving files and to allow for any number of subfolders, you need to create a special route. For example, a typical request for a file might look like this:

```
http://localhost:6234/Media/somefolder/somefile.html
```

To accomplish this, add the following route before the default route in the RouteConfig.cs file, as shown in Figure 2.

```
Figure 1 The web.config with custom handler for /Media/* path
```

```
Figure 2 The route to handle the /Media/* requests
```

The expression (*FilePath) tells ASP.NET MVC to handle any number of additional parameters as a single parameter named filePath.

Next, create a MediaController to handle the requests going to the /Media/* path, as shown in Figure 3.

```
<configuration>
 <system.webServer>
  <handlers>
   <add name="MediaWorkaround" path="/Media/*" type="System.Web.Handlers.TransferRequestHandler" />
  </handlers>
 </system.webServer>
</configuration>
```

```
<system.webServer>
  <handlers>
    <add name="MediaWorkaround" path="/Media/*" type="System.Web.Handlers.TransferRequestHandler" />
  </handlers>
</system.webServer>
```

Now you can test the configuration by inserting a breakpoint in the controller method, running the Web app, and navigating to any file under the Media folder. For example:

```
http://localhost:6234/Media/somefolder/somefile.html
```

CAN YOU TAKE
THE RISK?

Affordable High-Tech
Digital Escrow

TOWER 48 is the most advanced and affordable digital escrow solution available. Designed and built specifically for software and other digital assets, TOWER 48 makes escrow inexpensive and hassle free. Better yet, as a vendor, you can turn escrow into a service you offer to your customers and create a new revenue stream for yourself. Regardless of whether you are a vendor who wants to offer this service to your customers, or whether you are a customer looking for extra protection, visit our web site to start a free and hassle-free trial account or to learn more about our services and digital escrow in general!

Visit www.Tower48.com for more information!
tasks that have to be done... and they cannot be marched down or recast for these tasks. People are always ‘almost fits’ at best.” *(p55)* No matter who they are or what they know or how long they’ve been with the organization, they will always have weaknesses that need shoring up and/or a lack of knowledge that needs filling and/or any of a dozen other situations that require your time and attention. Some people will be free about admitting what they don’t know; some will see the admission of ignorance as a critical weakness; and some simply won’t realize that there are things they don’t know. It’s on you to make sure that all three categories of people are supported.

Summary

So, what now? “Time is precious, yeah, we get it. What do we do, then?”

For starters, admit that you probably don’t know where the time goes in your day. Start a journal or a time log. Record yourself or get an activity-tracking app for your laptop or mobile device that keeps a log of where you spend your day. I’ll bet you a nice bottle of wine that whenever you think you spend your time, there’s going to be something in that report that surprises you.

And, armed with this knowledge, start making more specific decisions. If you choose to play a video game, because you need to relax, then do so with finite limits put into place. (I allow myself one-co-op match on Starcraft II, for example, when I need to let my brain work on a problem sub-consciously.) If you want to watch TV, go for it! Nothing wrong with that. Just make sure you know when you’re going to stop. And so on. But be aware of how long you can shift resources from one area into another—in my house, we’ve had maids clean every week or so, because when I compare the cost of paying them double the wage against the time I (and my family) would have to spend doing it (and not as well), it’s a total no-brainer. Ditto for the gardener. There’s this one area where I can shift capital to spend time on other things.

Most of all, be very concerned with how your actions affect other peoples’ time. When setting up a meeting, does every single individual need to be there? Are there because they will be contributors, or because you just want them to know the meeting is about to happen? Could they get an hour back if you just sent them then a summary of the meeting afterwards? I routinely ignore come of all the mail I get. I use that hour or two to get other things done, and anything super-urgent that I need to know will inevitably filter back to me either directly or indirectly.

Time is a precious resource. Treat it with the same care and diligence you do your finances, if not more, and you will suddenly find that you are getting so much more out of it.
On Time

For an industry that prides itself on its analytical ability and abstract mental processing, we often don’t do a great job applying that mental skill to the most important element of the programmer’s tool chest—that is, ourselves. I have a friend who continuously mocks me for any response that contains the word “busy.” For the past year or so, she will test me, asking some question or remarking on some work idiom, and it may take me more than a day to reply. Almost without fail, I apologize for the late reply, citing some variation on the phrase “I had stuff to do that kept me too busy to respond.” Time, it seems, gets away from me on quite a regular basis.

Peter Drucker would have a field day with that paragraph.

Drucker on Time

“Effective executives know that time is the limiting factor. The output limited of any process are set by the scarcest resource. In the process we call ‘accomplishment,’ this is time.” (Effective Executive, p26)

It’s not like I don’t want to reply to my friend—she’s quite the interesting person to talk to, and we have some really good discussions about life, philosophy, technology, and a few other things. It’s just that there’s always something else to do: prepare for a conference talk, meetings with colleagues and/or future colleagues, write an article for CODE Magazine, dinner, shower, sleep… And I know I’m certainly not the only one that wrestles with this problem—how many times have you heard phrases like, “I can’t find the time” or “If there’s enough time” or “Time got away from me” or any of the hundreds of permutations on that theme? It’s almost like time is this elusive, mystical unicorn that only the most virtuous of us can find, capture, and enjoy.

“Time is also a unique resource. … The supply of time is totally inelastic. No matter how high the demand, the supply will not go up. There is no price for it and no marginal utility curve for it. Moreover, time is totally perishable and cannot be stored. Yesterday’s time is gone forever and will never come back. Time is, therefore, always in exceedingly short supply.” (p27)

We each of us receive exactly the same amount of time per day: 24 hours, 60 minutes per hour and 60 seconds per minute. There are numerous memes and motivational posters that will helpfully calculate what that all multiplies into, but the number isn’t the point: the point is that time, of all things, is the most egalitarian resource any of us will ever experience. No matter who you are, no matter where you were born, the color of your skin, or the balance in your account, you get exactly the same amount of time per day as the rest of us. And yet, for so many of us, that never seems to be enough. Where does that time go?

“Everything requires time. It is the one truly universal condition. All work takes place in time and uses up time. Yet most people take for granted this unique, irreplaceable, and necessary resource. Nothing else, perhaps, distinguishes effective executives as much as their tender loving care of time.” (p27)

A few years ago, when money was a little tight at the house, my wife and I decided that we were going to shut off regular TV service. This was actually before the heyday of streaming-media platforms that the world enjoys now, so we were making a pretty conscious decision to not have access to a lot of the normal programming that cable subscribers enjoy. We still had the TV itself, of course—we hooked up various console gaming devices to it so that the four of us in the family could spend time collectively tuned in to a video or DVD or game. (Mary is by far one of the best movie non-intense games ever made, and that’s the hill I’ll die on if I must.)

What we found is that turning off the TV opened up a whole range of possibilities that we hadn’t really spent much time exploring, both collectively and individually. We dove much deeper into board games for our time together at the house. My Kindle exploded with a ton of books, fiction and non—, that I now have with me. Even as I write this, there’s no TV anywhere in the house—everyone is engaged in some other pursuit, even video-gaming, that requires more attention and interaction than just drooling in front of the TV.

Turning off the TV told us that a lot of our time was going to watching TV. Where do you spend your time?

Spending time

To be clear, you don’t have to be an effective executive to understand time and its impact on your day. Drucker wrote, “... [Most of the time the executive require, for minimum effectiveness, a fairly large quantum of time. To spend in one stretch less than this minimum is sheer waste. One accomplishes nothing and has to begin all over again. ... To write a report may, for instance, require six or eight hours, at least for the first draft. It is pointless to give seven hours to the task by spending fifteen minutes twice a day for three weeks. All one has at the end of the is blank paper with some doodles on it.” (p30) Drucker intuitively understood the concept of flow, and software developers almost universally intuitively understand that as well. “The same goes for an experiment. One simply has to have five to twelve hours in a single stretch to set up the apparatus and to do at least one completed run. Or one has to start all over again after an interruption.” (p30) Substitute “experiment” and “apparatus” with “feature” and “environment,” and that sentence could easily appear in any developer’s handbook or blog post.

Drucker doesn’t just end there, though: He also knew that people require time, too. I can’t have a fire connection with my colleagues or co-workers without directives or managers if I can’t spend a minimum quantum of time with them. “The manager who thinks he can discuss the plans, direction, and performance of one of his subordinates in fifteen minutes is just deceiving himself. If one wants to get to the point of having an impact, one needs probably at least an hour and usually much more. And if one has to establish a human relationship, one needs infinitely more time.” (p31)

If your job is to manage people—and that is the job of managers, after all, foremost among all other tasks—then there needs to be a steady, deliberate and consistent investment of time on your part. “People-decisions are time-consuming, for the simple reason that the Lord did not create people as ‘resources’ for organisation. They do not come in the proper size and shape for the
ReSharper

The legendary extension to Visual Studio

jetbrains.com/resharper