ISSUE 22 | JAN - FEB 2016

DNCMAGAZINE

www.dotnetcurry.com

ASP.NET **WebHooks**

Process Customization in **Visual Studio Team Services**

Overview of ASP.NET 5

Biological modeling with AngularJS

Create your own Project Template for Visual Studio

Seeds: Liskov Substitution Principle

HomePi

A Windows 10 IoT app

CONTENTS

EDITORIAL

CREDITS

Editor In Chief
Suprotim Agarwal
suprotimagarwal@
a2zknowledgevisuals.com

Art Director
Minal Agarwal
minalagarwal@
a2zknowledgevisuals.com

Contributing Authors
Craig Berntson
Damir Arh
Gil Fink
Ravi Kiran
Shoban Kumar
Subodh Sohoni

Technical Reviewers
Ravi Kiran
Rion Williams
Suprotim Agarwal

Next Edition
3rd March 2016

POWERED BY
a2z Knowledge Visuals

Editor in Chief

Dear Readers, wish you all a Very Happy New Year 2016 and welcome to the 22nd Edition.

2015 was a remarkable year for Microsoft as it embraced Open Source, and incorporated today's open technologies and environments in its tools and services. As a result, not only is .NET cross-platform now for Linux and OSX; we also have free tools like Visual Studio Community and Code to create apps that can target every major device and OS.

2015 was a big year for JavaScript as well. ES6 was released in 2015 and ES7 is already in progress, Newer frameworks like Vue, Aurelia and React gained momentum. Ember, Knockout and Backbone continued to be popular and Angular was in news for its upcoming 2.0 release, which did give TypeScript a lot of attention.

In 2016, we will continue to invest our time to to help you to learn, prepare and stay ahead of the curve. Cheers to learning!

Feel free to email me at **suprotimagarwal@dotnetcurry.com**

Suprotim Agarwal

.NET & JavaScript Tools

Shorten your Development time with this wide range of software and tools

CLICK HERE

ASP.NET MVC CONTROLS

WORK EFFORTLESSLY WITH ASP.NET MVC

Quickly create advanced, stylish, and high performing UIs for ASP.NET MVC with Ignite UI MVC. Leverage the full power of Infragistics' JavaScript-based jQuery UI/HTML5 control suite with easy-to-use ASP.NET MVC helpers and get a jump start on even the most demanding Web applications.

Download ASP.NET MVC Controls as part of the Ultimate Developer toolkit.

DOWNLOAD FREE TRIAL

INFRAGISTICS®

ASP.NET WebHooks

The ASP.NET web stack provides solutions to most of the problems we face while building server-based web applications. It has support for building Web applications with views using Web Forms and MVC, building REST APIs using Web API, and support for real-time communication using SignalR. The platform continues to grow with each of these supported technologies receiving frequent updates to make each of them even better. A recent addition to the ASP.NET stack is WebHooks. In this article, we will see what WebHooks are, examine their usefulness, and we will implement an application consuming WebHooks from GitHub.

*A special note of thanks to **Rion Williams** for reviewing the article and adding important bits to it.*

What are WebHooks?

WebHooks are user-defined HTTP-based callbacks. They are configured with an event so that the callback will be invoked when the triggering event occurs. These callbacks can be configured and managed by both users of the application (including third-parties) and developers. The source of these events may be in the same application or, in a different application.

The REST API is called when the triggering event occurs and then data related to the event is sent to the POST API. Basically, a user or developer can create a notification and when a specific trigger occurs, they will be notified via an HTTP POST. Web Hooks are most commonly used in systems such as continuous integration systems or messaging queues; but because of their flexibility (events and callbacks can be just about anything), their potential could really be endless.

ASP.NET WebHooks

ASP.NET WebHooks is the implementation of WebHooks in the context of ASP.NET. This feature is currently implemented in ASP.NET 4.5 and it is in preview at the time of writing this article. ASP.NET WebHooks supports both sending and receiving WebHooks. It uses ASP.NET Web API for sending and receiving WebHooks under the covers.

Online services like GitHub, Trello, Dropbox, Slack and a few others expose WebHooks to notify actions. ASP.NET team has implemented a set of NuGet packages to make it easier to talk to these services. It is also possible to subscribe to other services using the custom receiver NuGet package.

In this article, we will see how this feature can be used to receive WebHooks exposed by Github. Before we start writing code, we need to understand how this communication is designed to work. Figure-1 explains the flow of data when a WebHook is raised.

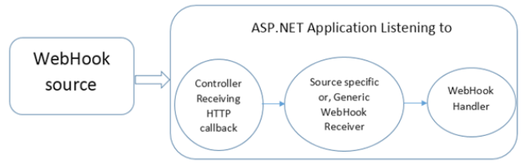


Figure 1: Flow of Data when a WebHook is raised

When a WebHook is raised by a source,

- The source calls the receiver's HTTP POST endpoint
- The Web API controller handling the HTTP POST callback forwards the notification to the receiver specific to the sender or, to the generic receiver
- The receiver validates the request and then sends it to the WebHook handler

WebHook handler is the component that gives us access to the data received from the source. Then it is responsibility of the application to use this data.

The data may be stored in a database, may be sent to the client and displayed on the UI or, may be logged on the server based upon the need of the application.

Implementing a GitHub WebHook Client using ASP.NET WebHooks

Now that we have some understanding on what WebHooks are and how they work in ASP.NET, let's build an application to understand how WebHooks from GitHub can be received in an ASP.NET application.

Setting up the Application

Open Visual Studio 2015 or, 2013 and create a new empty ASP.NET 4.5 application. To make it a bit easier to setup, **choose the Web API checkbox** in the New ASP.NET Project dialog.

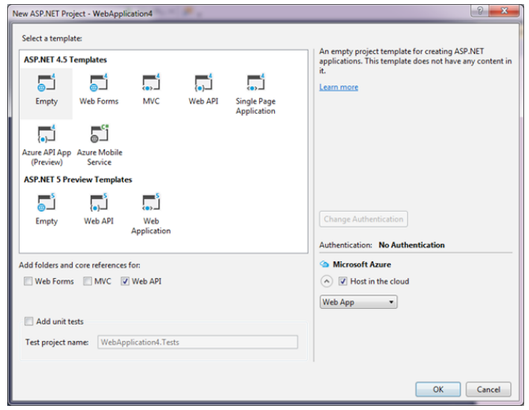


Figure 2: Empty ASP.NET Project

As most of the WebHook providers are hosted on HTTPS and they expect the receivers also to use HTTPS protocol, host this application on Azure to get SSL for free. In addition to this, since only a public domain can ping back another public domain, hosting on Azure makes our app public. **In the Azure Web App hosting dialog box, enter URL of the application and hit the OK button**.

Now we need to add WebHooks to this application.

This can be done in one of the two ways:

a) installing the Nuget packages (Microsoft.AspNet.WebHooks.Common, Microsoft.AspNet.WebHooks.Receivers and other specific receiver packages), configuring the WebHook receiver and manually writing the WebHook handler OR

b) taking advantage of the WebHooks extension. Brady Gaster has a great video in the announcement blog post of the extension demonstrating usage of the extension. For the purposes of this article, we will be demonstrating the manual, NuGet approach.

Before we configure the project to use GitHub WebHooks, we need to configure WebHooks on a GitHub repo. As we need to make the site public, let's deploy the application on Azure. **Right-click on the project and choose the Publish option**. The dialog box will be prepopulated with the data using the Azure account associated with your instance of Visual Studio. After reviewing the information, **click the Publish button** to publish the site.

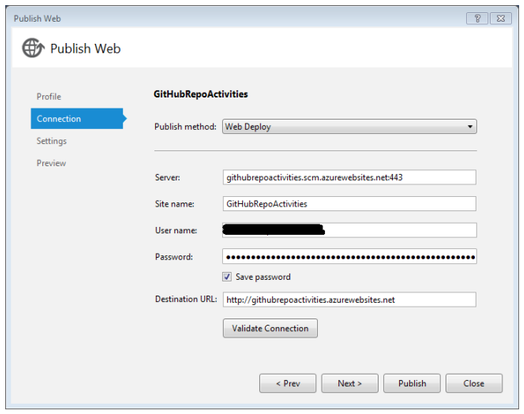


Figure 3: Deploy application on Azure

Now the site should be deployed on Azure and will be loaded into an instance of your default browser.

Setting up WebHooks on a GitHub Repo

Login to GitHub using your GitHub account credentials. If you don't already have an account on GitHub, then create one and login. Create a new repo under your account and give it a name of your choice. We will use this repo to play with GitHub's WebHooks.

Once the repo is created, **open the Settings page of the repo** and from the menu on the left side, **choose WebHooks and Services**. In this page, **click the "Add WebHook" button**. This will ask for password. Enter your password and now you will see a page asking for details of the WebHook. **Fill in the following details into the WebHook dialog**:

- Payload URL: Payload URL for GitHub would follow this pattern: https://your-site-name.azurewebsites.net/api/webhooks/incoming/github
- Content type: application/json
- Secret: You can generate a secret using the Free Online HMAC Generator
- Which events would you like to trigger this webhook? : Send me everything

Click the "Add Webhook" button after filling these details. This will create a WebHook on GitHub. It will send a WebHook to the site as soon as it is created, but it will fail as our server is not yet ready to listen to this WebHook. Let's make our server to listen for this event and capture the WebHooks.

Creating GitHub WebHooks Receiver

To enable listening to GitHub WebHooks, we need to **install the following NuGet package in the project by typing the following into the Package Manager Console**:

> Install-Package Microsoft.AspNet.WebHooks.Receivers.GitHub –pre

Note: *The package can also be installed from the NuGet Package Manager GUI by searching for the appropriate package.*

As the package is still in prerelease state (as of this writing), we need to include the –pre option in the command. The above package brings the packages *Microsoft.AspNet.WebHooks.Receivers and Microsoft.AspNet.WebHooks.Common*, as it depends on them.

Open the Web.config file and add the following entry under the appSettings section:

```
<add key="MS_WebHookReceiverSecret_
GitHub" value="your-github-secret" />
```

The "your-github-secret" value in the above entry should be same as the secret entered previously on the GitHub site. Now we need to enable WebHooks in the current project. For this, **add a new C# class file to App_Start folder and name it *WebHookConfig*. Open this file and add the following code to it:**

```
public static class WebHookConfig
{
  public static void
  Register(HttpConfiguration config)
  {
    config.
    InitializeReceiveGitHubWebHooks();
  }
}
```

We need to invoke this method when application starts to activate WebHooks. **Open Global.asax.cs and add the following statement to *Application_Start* event:**

```
GlobalConfiguration.
Configure(WebHookConfig.Register);
```

Publish the project again after making the above change. Now we have WebHooks active and running on the site. **Go to GitHub, open the project for which WebHook is configured and change the readme file using the edit option on GitHub and commit it. This should generate a WebHook**. You can check status of the WebHook under the same page where the WebHook was configured. It should show a succeeded WebHook delivery similar to the one in Figure 4:

Figure 4: A WebHook generated and received

Creating a WebHook Handler

The screenshot in Figure 4 indicates that the WebHook is received by the Web API in our application. The Web API then looks for presence of WebHook handlers in the application. If it finds them, the information received is sent to the handler for further processing. If the application has to use the data received, then WebHook handler is the way to go.

Add a new folder to the application and name it WebHookHandlers. Add a new C# class file to this folder and name it GitHubWebHookHandler. Add the following code to this file:

```
public class GitHubWebHookHandler :
WebHookHandler
{
  public override Task
  ExecuteAsync(string receiver,
  WebHookHandlerContext context)
  {
    if ("GitHub".
    Equals(receiver,StringComparison.
    CurrentCultureIgnoreCase))
    {
      string action = context.Actions.
      First();
      JObject data = context.
      GetDataOrDefault<JObject>();
      var dataAsString = Newtonsoft.
      Json.JsonConvert.
      SerializeObject(data);

    }

    return Task.FromResult(true);
  }
}
```

In the snippet you just saw:

- The class *GitHubWebHookHandler* is extended from the abstract *WebHookHandler* class. *WebHookHandler* is the contract defined by WebHooks and any class extending this abstract class is instantiated as soon as a WebHook is received

- It overrides the abstract *ExecuteAsync* method. This method gets access to name of the receiver and the *WebHookHandlerContext* object. The context object contains the actions that caused the WebHook, ID of configuration of the WebHook and the data received by the WebHook

The data in the context object has the type *object*. We need to convert it to a type of our choice before using it. Out of all possible activities on a GitHub repo, I chose to handle commits, opening issues, closing issues and commenting on an issue. **After these changes, Build and publish the application to Azure.**

Connect to your Azure account within the Server Explorer of Visual Studio and **start debugging the application using the Attach Debugger** option.

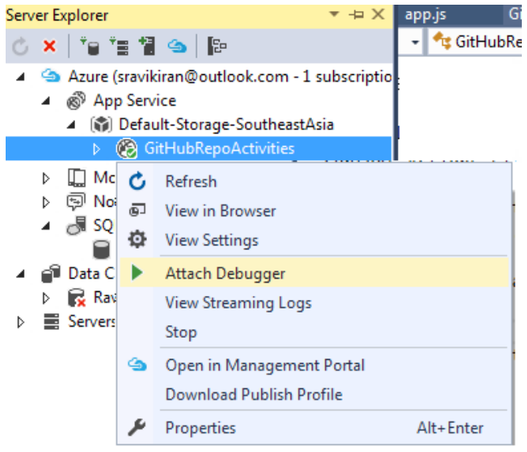


Figure 5: Preparing to debug the application

Place a breakpoint inside the *if* block of the code snippet below and perform a commit operation on the GitHub repo. A WebHook will be sent by GitHub to the application about this operation and we will be able to see details of the action if you inspect *data* object in the snippet:

Figure 6: Inspecting data object

Similar objects would be sent for other operations as well, their structure would differ as they hold different data. In the next section, we will store selected values from this payload in the database.

Saving the Data Received in a Database

To save the data received, **create a database on SQL Azure**. Connect to this database using either SQL Server Management Studio or, using SQL Server tools on Visual Studio. **Run the following query on this database to create a table where we will save the data:**

```
CREATE TABLE GitHubActivities
(
  ActivityId INT IDENTITY PRIMARY KEY,
  ActivityType VARCHAR(20),
  [Description] VARCHAR(MAX),
  Link VARCHAR(MAX)
);
```

The table would contain type of the event, the description (commit message in case of commits, title of the issue when an issue is opened or closed and text of the comment when a comment is posted) and a URL of the page where the event can be seen. We need to transform the data received from GitHub to this form before saving. We will do it in the next section.

Add a new ADO.NET Entity Franework Data Model to the Models folder and name it *GitHubActivityEntities*. Configure this model to point to the SQL Azure database we created for this application and choose the GitHubActivities table in the wizard.

We need to perform two types of GET operations, one to fetch all activities and the other to fetch activities that were posted after a given activity id. In addition to these, we need to perform an insert operation to insert details of the new activities. The following is the repository class that performs these operations:

```
public class GitHubActivitiesRepository:
IDisposable
{
  RaviDevDBEntities context;
  public GitHubActivitiesRepository()
  {
    context = new RaviDevDBEntities();
  }
  public List<GitHubActivity>
       GetAllActivities()
  {
    return context.GitHubActivities.
         ToList();
  }
  public List<GitHubActivity>
            GetActivitiesAfter(int
activityId)
  {
    return context.GitHubActivities.
         Where(gha => gha.ActivityId >
              activityId).ToList();
  }
  public GitHubActivity
       AddNewActivity(GitHubActivity
activity)
  {
    var activityAdded = context.
                GitHubActivities.
Add(activity);
    context.SaveChanges();

    return activityAdded;
  }

  public void Dispose()
  {
    context.Dispose();
  }
}
```

We will invoke the GET operations from a Web API controller to make the data retrieval possible from JavaScript and the add operation will be performed from the *GitHubWebHookHandler* class. Before calling the add method, we need to convert data to *GitHubActivity* type. The data received from GitHub differs for every event and we need to understand the pattern before converting. After a few observations, I understood that we need an intermediate object to help us in this scenario. After some analysis, I created the following class to hold the required information from the payload we receive and the enum to represent type of the event:

```
public class GitHubMessage
{
  public JObject[] Commits { get; set; }
  public JObject Issue { get; set; }
  public JObject Comment { get; set; }
  public string Action { get; set; }
  public GitHubActivityType ActivityType
  { get; private set; }

  }
  public enum GitHubActivityType
  {
    Commit,
    IssueOpened,
    IssueClosed,
    Comment
  }
```

The payload would contain some of the properties of the class and we will use JSON.NET to convert the data. The value of the property *ActivityType* has to be calculated based on values in the other properties. **Add the following method to the GitHubMessage class to find the *ActivityType*:**

```
public void SetActivityType()
{
  //If the payload has commits, it is a
  commit
  if (Commits != null && Commits.Length
  > 0)
  {
    ActivityType = GitHubActivityType.
    Commit;
  }
  //if the payload has Issues property
  set and Action is set to opened, an
  issue is opened
  else if (Issue != null &&
  string.Equals("OPENED", Action,
  StringComparison.
  CurrentCultureIgnoreCase))
  {
    ActivityType = GitHubActivityType.
    IssueOpened;
  }
  //if the payload has Issues property
  set and Action is set to closed, an
  issue is closed
  else if (Issue != null       &&
```

```
  string.Equals("CLOSED", Action,
  StringComparison.
  CurrentCultureIgnoreCase))
  {
    ActivityType = GitHubActivityType.
    IssueClosed;
  }
  //if the payload has comment set, it
  is a comment
  else if(Comment != null      &&
  string.Equals("CREATED", Action,
  StringComparison.
  CurrentCultureIgnoreCase))
  {
    ActivityType = GitHubActivityType.
    Comment;
  }
}
```

Comments in the snippet explain how the *ActivityType* is assigned. Now that we have the type, we need to find *Description* and *Link* properties of the *GitHubActivity* class and assign them to an object. The following method does this:

```
public GitHubActivity
ConvertToActivity() {
 var activity = new GitHubActivity();
 activity.ActivityType = ActivityType.
ToString();

 if (ActivityType ==
GitHubActivityType.IssueClosed ||
ActivityType == GitHubActivityType.
IssueOpened) {
  activity.Link = Issue.
  GetValue("html_url").Value<string>();
  activity.Description = Issue.
  GetValue("title").Value<string>();
 }
 else if (ActivityType ==
 GitHubActivityType.Commit)
 {
  activity.Link = Commits[0].
  GetValue("url").Value<string>();
  activity.Description = Commits[0].
  GetValue("message").Value<string>();
 }
 else
 {
  activity.Link = Comment.
  GetValue("html_url").Value<string>();
  activity.Description = Comment.
  GetValue("body").Value<string>();
 }
 return activity;
}
```

These methods have to be called from the GitHubWebHookHandler class and the final result has to be used to call add() method on the repository class. Modify the class GitHubWebHookHandler as follows:

```
public class GitHubWebHookHandler :
WebHookHandler {
 GitHubActivitiesRepository repository;

 public GitHubWebHookHandler()
 {
  repository = new
  GitHubActivitiesRepository();
 }

 public override Task
 ExecuteAsync(string receiver,
 WebHookHandlerContext context)
 {
  if ("GitHub".
  Equals(receiver,StringComparison.
  CurrentCultureIgnoreCase))
  {
   string action = context.Actions.
   First();
   var message = context.
   GetDataOrDefault<GitHubMessage>();
   message.SetActivityType();

   var activityToSave = message.
   ConvertToActivity();
   repository.
   AddNewActivity(activityToSave);
  }

  return Task.FromResult(true);
 }
}
```

Save all files, build and publish the application and then perform some operation on the repo. You will immediately see the data getting added to the table on SQL Azure.

Building a Web API Controller and a Page to View Activities

Now that we have data flowing into our database whenever an action takes place, let's view this data on a page. Let's build an ASP.NET Web API controller to serve this data. As already discussed, the controller will have APIs to serve all records from the table and a set of records after a certain activity.

Add a new Web API controller to the Controllers folder and name it *ActivitiesController* and add the following code to it:

```
public class ActivitiesController :
ApiController {
 GitHubActivitiesRepository repository;

 public ActivitiesController()
 {
  repository = new
  GitHubActivitiesRepository();
 }

 public IEnumerable<GitHubActivity>
 GetAllActivities()
 {
  return repository.GetAllActivities();
 }

 public IEnumerable<GitHubActivity>
 GetActivitiesAfter(int id)
 {
  return repository.
  GetActivitiesAfter(id);
 }
}
```

Add a new HTML file to the project and name it Index.html. Add references of jQuery library and Bootstrap's CSS file to this page. It is up to you if you want to add these references from CDN or, using bower. We will have a table displaying the list of activities on this page and the table will be updated after every two minutes to show the latest activities. **Copy following markup into body section of the page:**

```
<div class="container">
 <div class="row text-center">
  <h1>GitHub Activities using WebHooks
  </h1>
  <hr />
  <div class="col-md-10">
   <table class="table"
   id="tblGithubActivities">
    <thead>
     <tr>
      <th style="text-align:
      center">Activity Type</th>
      <th style="text-align:
      center">Description</th>
      <th style="text-align:
      center">Check on GitHub</th>
     </tr>
    </thead>
   </table>
  </div>
 </div>
</div>
```

Finally, we need some JavaScript to call the API and fill the table with rows. It also has to keep checking for new activities at an interval of two minutes and add the new entries to the table. **Add a new JavaScript file app.js to the application and paste following code in it:**

```
(function () {
 $(function () {
  var table = $("#tblGithubActivities");
  var lastActivityId;

  loadActivities().then(appendRows);

  setInterval(function () {
   loadActivitiesAfter(lastActivityId).
   then(function (data) {

    if (data.length > 0) {
     appendRows(data);
    }
   });
  }, 2*60*1000);

 function appendRows(data) {
  lastActivityId = data[data.length-1].
  ActivityId;

  var rows = "";
  data.forEach(function (entry) {
   rows = rows + "<tr><td>" + entry.
   ActivityType + "</td><td>" + entry.
   Description + "</td><td><a href='" +
   entry.Link + "'>Check on GitHub</a>
   </td></tr>";
  });
  table.append(rows);
 }
});

function loadActivities() {
 return $.get('/api/activities');
}

function loadActivitiesAfter(activityId)
{
 return $.get('/api/activities/' +
 activityId);
} })();
```

Save all the files, build and publish the application.

Now you will be able to see a page similar to Figure – 7.

GitHub Activities using WebHooks

Activity Type	Description	Check on GitHub
IssueClosed	Improve readme	Check on GitHub
Commit	Update README.md	Check on GitHub
IssueOpened	Improve readme	Check on GitHub
Comment	Improved!	Check on GitHub
Commit	To kick webhook	Check on GitHub
IssueClosed	Improve readme	Check on GitHub
Commit	Second line added	Check on GitHub
Commit	Formatted second line	Check on GitHub
Commit	Added third line	Check on GitHub
Comment	Really improved?	Check on GitHub

Figure 7: Github Activities using WebHooks

Conclusion

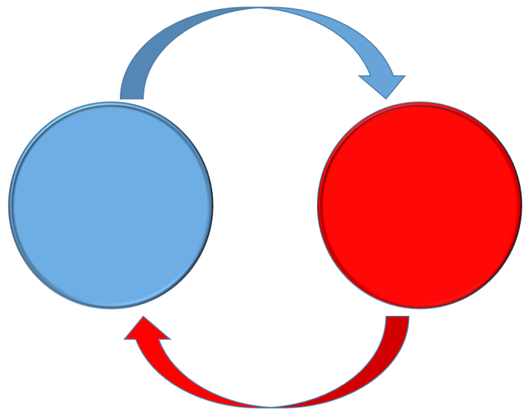
WebHooks provide a nice and simple way to keep monitoring activities on an online service. The data collected from WebHooks from certain services can even be used to take some crucial business decisions or, build a notification system on top of it. ASP.NET WebHooks provides abstractions on top of most widely used services and it has extensibility points to define our own senders and receivers. We will look at more capabilities of this feature in a future post ■

Download the entire source code from GitHub at bit.ly/dncm22-aspnet-webhook

About the Author

Rabi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

> I'm at the half-way point in my writing about SOLID technics. This issue, I look at the Liskov Substitution Principle (LSP). This is the only SOLID principle named after the person that originally promoted the concept.



SEEDS:

LISKOV SUBSTITUTION PRINCIPLE

In 1998, Barbara Liskov, put forth this principle in SIGPLAN Notices. Here's what she said,

"What is wanted here is something like the following substitution property: If for each object o2 of type T such that for all programs P defined in terms of T, the behavior or P is unchanged when o1 is substituted for o2 then S is a subtype of T."

Editorial Note:

You can read about Single Responsibility Principle and Open-Closed Principle over here http://www.dotnetcurry.com/software-gardening/1148/solid-single-responsibility-principle and http://www.dotnetcurry.com/software-gardening/1176/solid-open-closed-principle .

What Barbara Liskov specifically meant by this has been debated for years. You can follow some of this at http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple, where Robert "Uncle Bob" Martin, Alistair Cockburn, Michael Feathers, and others discuss the real meaning of LSP and even if LSP is a real OOP principle. I'm not going to debate whether LSP is or is not a real principle of Object Orientation. I'm going to follow the general belief that it is. And I'm going to give you the generally accepted definition.

> Uncle Bob summarized LSP as
>
> *Subtypes must be substitutable for their base types.*

In other words, given a specific base class, any class that inherits from it, can be a substitute for the base class. The classic example of this shows how you cannot substitute a rectangle class for a square class.

```
class LSPDemo
{
  static void Main(string[] args)
  {
    Rectangle shape;
    shape = new Rectangle();
    shape.SetWidth(14);
    shape.SetHeight(10);
    Console.WriteLine("Area={0}", shape.
    Area); // 140

    shape = new Square();
    shape.SetWidth(14);
    shape.SetHeight(10);
    Console.WriteLine("Area={0}", shape.
    Area); // 100
    Console.ReadLine();
  }
}

public class Rectangle
{
  protected int _width;
  protected int _height;

  public int Width
  {
    get { return _width; }
  }

  public int Height
  {
    get { return _height; }
  }

  public virtual void SetWidth(int
  width)
  {
    _width = width;
  }

  public virtual void SetHeight(int
  height)
  {
    _height = height;
  }

  public int Area
  {
    get { return _height * _width; }
  }
}

public class Square : Rectangle
{
  public override void SetWidth(int
  width)
  {
    _width = width;
    _height = width;
  }

  public override void SetHeight(int
  height)
  {
    _width = height;
    _height = height;
  }
}
```

Running this code will result in the first shape having an area of 140 and the second, an area of 100. You'd then be tempted to fire up the debugger

to figure out what's going on. The behavior of the subclass has changed. Using Uncle Bob's words, the code violates LSP since subtype is not substitutable for its base type. Now, we all know that a square and rectangle are not the same, but this shows that a square is not a special type of a rectangle.

This is where most discussions of LSP stop. But LSP is more complex than this brief example. There are several rules that LSP actually enforces. These rules fall into two categories, contract rules and variance rules. Let's make a more in depth examination of these rules.

Contract rules

When creating a class, the contract states in formal terms how to use the object. You expect the name of the methods and the parameters and data types of those parameters as input, and then what data type to expect as a return value. LSP places three restrictions on contract rules:

- **Preconditions cannot be strengthened by the subtype** – Preconditions are the things required by the method to run reliably. This would require the class get instantiated correctly and required parameters are passed to the method. Typically, guard clauses are used to enforce that parameters are of the correct values.

- **Postconditions cannot be weakened in the subtype** – Postconditions verify the object is left in a reliable state when the method returns. Guard clauses are again used to enforce postconditions.

- **Invariants of the supertype must be preserved by the subtype** – Invariants are things that must remain true during the lifetime of the object once object construction is finished. This could be a field value that gets set in the constructor and is assumed to not change. The subtype should not change these type of fields to invalid values. For example, there may be a business rule that minimum shipping charge field must be greater than or equal to zero. The subtype should not make it negative. Read-only fields guarantee this rule is followed.

Variance rules

Before explaining variance rules, we need to define variance.

Here's what Wikipedia says,

> *Variance refers to how subtyping between more complex types relates to subtyping between their components…The variance in a C# interface is determined by in/out annotations on its type parameters.*

https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science). That was rather complicated. Think of it like this, how do you expect different, but related subtypes to behave? That is variance. Now, here are the LSP variance rules:

- There must be a contravariance of method arguments in the subtype – The subtype reverses the ordering of types.

- There must be covariance of return types from the method in the subtype – The subtype keeps types in the same order, from specific to most generic.

- The subtype should not throw new exceptions unless those exceptions are subtypes of exceptions thrown by the base type – This rule is easy and self-explanatory.

If you extrapolate these rules, you find that a subtype cannot accept a more specific type as an argument and cannot return a less specific type as the result.

A solution to Liskov

Now that I've shown what LSP is not and explained what it is, I would like to show you an example of what the Liskov Principle is. The problem is, we need to substitute square for rectangle, or vice-versa, with only changing the class that gets instantiated. But, when creating a square, if we pass two parameters for height and width, which parameter is the correct one? We could require that you pass the same number twice, but that seems rather useless as only one is needed. What this comes down to is that the above code is good to show what Liskov is not, but doesn't work well when you try to show what Liskov is. So, I have to turn to a different example.

I've decided to use a base type of Animal. Granted, not a perfect example as some animals hop, others slither, walk, or gallop. Some fly, others don't. Some have feet, others don't. But I think it will still show how to substitute one subclass for another.

```
class Program
{
  static void Main(string[] args)
  {
    Animal animal = new Dog();
    Console.WriteLine(animal.Walk());
    Console.WriteLine(animal.Run());
    Console.WriteLine(animal.Fly());
    Console.WriteLine(animal.
    MakeNoise());
    Console.ReadLine();
  }
}

public class Animal
{
  public string Walk()
  {
    return "Move feet";
  }

  public string Run()
  {
    return "Move feet quickly";
  }

  public virtual string Fly()
  {
    return null;
  }

  public virtual string MakeNoise()
  {
    return null;
  }
}

public class Dog : Animal
{
  public override string MakeNoise()
  {
    return "Bark";
  }
}

public class Bird: Animal
{
  public override string MakeNoise()
  {
    return "Chirp";
  }

  public override string Fly()
  {
    return "Flag wings";
  }
}
```

If you run this code for Dog, you get the following output.

```
Move feet
Move feet quickly

Bark
```

Note the blank line for Fly( ). Dogs don't fly, so nothing gets returned. Now change Dog to Bird and you get

```
Move feet
Move feet quickly
Flag wings
Bark
```

What happens if you now change Bird to Animal?

```
Move feet
Move feet quickly
```

A subclass (Dog or Bird) can be substituted for the base class (Animal) and everything still works. The code itself doesn't care. We could also take this one step further and use a Factory method to create the class we want to use, and the code wouldn't even know the class at all. But that's beyond this discussion.

So now you know about Liskov Substitution Principle. It's well defined rules for using subtypes in place of the base type. By following these rules, and others in SOLID, you will have better software that is more maintainable, easier to extend, and less fragile. Your software garden will be lush, green, and thriving.

About Software Gardening

Comparing software development to constructing a building says that software is solid and difficult to change. Instead, we should compare software development to gardening as a garden changes all the time. Software Gardening embraces practices and tools that help you create the best possible garden for your software, allowing it to grow and change with less effort ■

About the Author

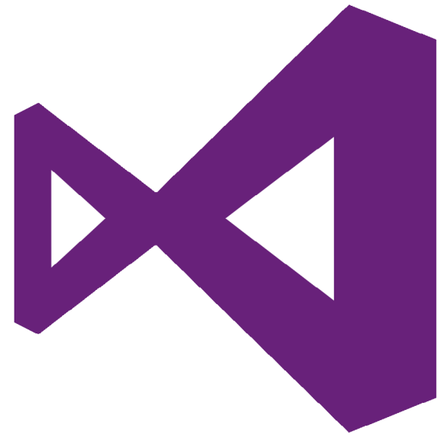
craig berntson

Microsoft® Most Valuable Professional

Craig Berntson is the Chief Software Gardener at Mojo Software Worx, a consultancy that specializes in helping teams get better. He has spoken at developer events across the US, Canada, and Europe for over 20 years. He is the co-author of "Continuous Integration in .NET" available from Manning. Craig has been a Microsoft MVP since 1996. Email: craig@mojosoftwareworx.com, Blog: www.craigberntson.com/blog, Twitter: @craigber. Craig lives in Salt Lake City, Utah.

Usually, there is no need to create your own Visual Studio project template. The existing ones should suffice for most cases. However, if you find yourself often creating new projects and manually adding the same files or references to them; creating a new project template might make sense. It will save you time and prevent minor variations between the projects.

CREATE YOUR OWN PROJECT TEMPLATE FOR VISUAL STUDIO 2013 AND 2015



Preparing the Desired Project

The starting point for any new project template is another Visual Studio project, structured exactly as you would like a new project to look like, when you create it. As an example, we will prepare a new class library project, which you could use for trying out small code snippets to answer questions in forums or solve programming exercises. It will contain two classes: one for the sample code and one for the tests. NUnit unit testing framework will be referenced as a NuGet package.

I will be using Visual Studio 2015 for all my samples. Unless mentioned otherwise, the experience is the same in Visual Studio 2013. Just replace 2015 with 2013 wherever it appears in the text. You can use Visual Studio Community Edition or any of the other editions, except Express.

Here are the steps for creating such a project:

- Create a new classic Windows desktop **Class Library** project and name it **Sample.Project**. Keep the option checked to create a directory for the solution.

- Delete the **Class1.cs** file that was added to the project automatically.

- Open the **Manage NuGet Packages** window for the project, search for **NUnit** and install its latest stable version. Note that the user interface in Visual Studio 2013 slightly differs from the screenshot below.

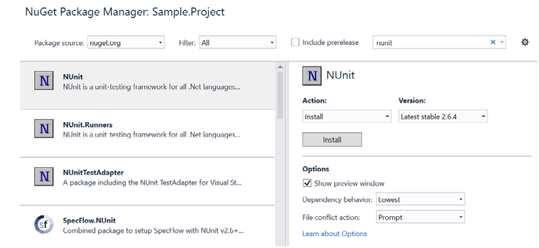


Image 1: Install NUnit NuGet package

- Add a new class item named **Sample.cs** with the following code:

```
namespace Sample.Project
{
  class Sample
  {
    public int Run(int[] args)
    {
      return 0;
    }
  }
}
```

- Add another class item named Tests.cs with the following code:

```
using NUnit.Framework;

namespace Sample.Project
{
```

```
  [TestFixture]
  class Tests
  {
    [Test]
    public void SampleData()
    {
      Test(null, 0);
    }

    private void Test(int[] input, int
    expected)
    {
      var sample = new Sample();
      var actual = sample.Run(input);
      Assert.AreEqual(expected, actual);
    }
  }
}
```

Build the project and run the test (you will need NUnit Test Adapter or a similar extension to run a NUnit based test inside Visual Studio). If it passes, the project is correctly configured and we can use it to create a project template from it.

Exporting the Project as a Template

The easiest way to create a project template is by using the **Export Template Wizard** that is built into Visual Studio. Just navigate to **File > Export Template...** to open it. On the first page you can choose between creating a project template or an item template (**Project template** in our case), and select the project you want to create the template from (**Sample.Project** in our case).

On the second page, you will have a chance to enter the name and description of the template, and select the optional icon and preview image. You can keep both options checked to automatically import the template (i.e. copy it to **Visual Studio 2015\Templates\ProjectTemplates** inside My Document folder) and open the output directory (as shown in the wizard).

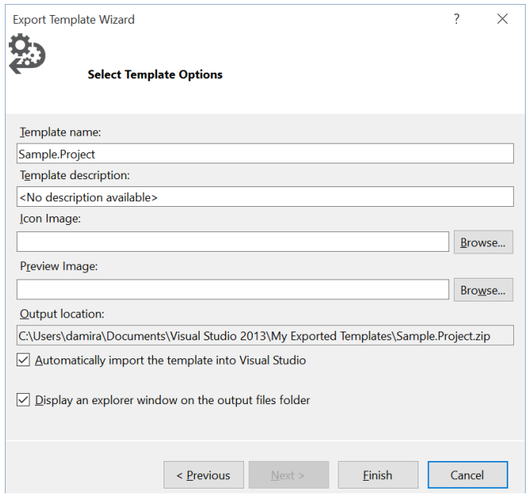


Image 2: Enter project template details

If you are not too demanding, your job is already done. The exported template is available when creating a new project in Visual Studio. If chosen, it will create the same initial structure as you have prepared it. If you are using NuGet 2.7 or newer, the NuGet package will automatically be downloaded before the build and the test will succeed when run. You can even send the generated Sample.zip file to other developers. As long as they copy it into Visual Studio 2015\Templates\ProjectTemplates subfolder of My Documents, it will work for them, as well.

This approach has a couple of drawbacks:

- You cannot publish a template packaged as a .zip file to Visual Studio Gallery, to make it publically available.

- There is no installer for the template; each developer must manually copy it to the right directory to install or update it.

- If you want to do any changes to the template, you need to change your source project and repeat the export procedure.

- You cannot customize the project creation in any way, except for the automatic namespace and assembly name changes in the included source code files.

To avoid these downsides, you will need to take the longer route and create two more projects in your solution: a project template project to customize the created template and a VSIX project to distribute it as a Visual Studio extension. Both of them are available inside Visual Studio SDK.

If you are using Visual Studio 2015, Visual Studio SDK is a part of its setup package, but not installed by default: you must manually check the Visual Studio Extensibility feature during install. If you have not done so already, the only "project template" in Visual C# > Extensibility node will be **Install Visual Studio Extensibility Tools**. By double-clicking it, you will trigger the installation of the missing feature.

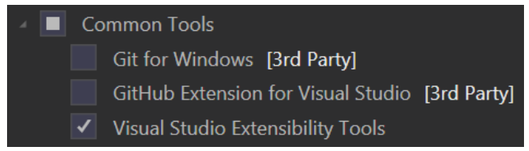


Image 3: Visual Studio Extensibility Tools feature in Visual Studio Setup

Visual Studio 2013 does not come bundled with its extensibility SDK. Instead, it is available as a standalone download that you need to install separately.

Creating a Project Template Manually

Once you have Visual Studio extensibility tools installed, there is a Visual Studio project template available for creating customized project templates without using the Project Template Wizard. It is named **C# Project Template** and can be found in the **Visual C# > Extensibility** node of Add New Project Dialog. Create a new project from this template in your solution and name it **Sample.Template**.

A couple of files inside the created project require a closer look.

There are two **AssemblyInfo.cs** files in this project. The one in Properties folder is a part of the template project and can be safely ignored. The one in the root folder will be included in the generated template, but since it extensively uses template parameters (keywords enclosed between two $ characters), there is usually no need to modify it:

```
[assembly:
AssemblyTitle("$projectname$")]

[assembly:
AssemblyCompany
("$registeredorganization$")]

[assembly: AssemblyProduct
("$projectname$")]

[assembly: AssemblyCopyright("Copyright
© $registeredorganization$ $year$")]

[assembly: Guid("$guid1$")]
```

A full list of supported template parameters is available in online reference documentation.

Class1.cs can be safely deleted, since we do not need it in our project. Instead, we need to add **Sample.cs, Tests.cs** and **packages.config**. I suggest you take them from Sample.zip created by the wizard and not from the initial project, because the fixed namespace is already replaced with $safeprojectname$ template parameter. Also, make sure you change the **Build Action** from Compile to **None** in the properties window for the .cs files, otherwise the build will fail because of invalid namespace name and other errors.

Sample.Template.vstemplate is the manifest file for the project template. It contains a list of all files that need to be included in the generated template; therefore, we need to change it in accordance with the changes that we have just made:

```
<TemplateContent>
 <Project File="ProjectTemplate.csproj"
 ReplaceParameters="true">

  <ProjectItem ReplaceParameters="true"
  TargetFileName="Properties\
  AssemblyInfo.cs">
   AssemblyInfo.cs
  </ProjectItem>
  <ProjectItem ReplaceParameters="true"
```

```
  OpenInEditor="true">
   Sample.cs
  </ProjectItem>

  <ProjectItem ReplaceParameters="true"
  OpenInEditor="true">
   Tests.cs

  </ProjectItem>
  <ProjectItem>packages.config
  </ProjectItem>
 </Project>
</TemplateContent>
```

We have replaced the entry for the removed `Class1.cs` with entries for the newly added files. Although I am not going to describe every supported feature of `TemplateContent` element, I think it is appropriate to mention the ones used in the snippet above:

- `ReplaceParameters` attribute enables the processing of template parameters (e.g. `$safeprojectname$`) in the file. If it is disabled, the file will end up in the project unchanged. Any file containing template parameters, must have this attribute set to `true`.

- `TargetFileName` attribute can be used to move or rename the file in the generated project. In our case, `AssemblyInfo.cs` is moved to `Properties` subfolder.

- `OpenInEditor` attribute indicates which files will automatically be opened when the project is created from the template: `Sample.cs` and `Tests.cs` in our snippet.

In `TemplateData` element, only the following values need to be changed to give your template proper identity; the rest you can leave unchanged:

```
<Name>Sample</Name>
<Description>Answer a forum question
or solve a programming exercise</
Description>
<DefaultName>Sample</DefaultName>
```

If you want to further customize the manifest file, you can take advantage of IntelliSense in the editor, which lists available elements, attributes and values. For more information, consult the online documentation.

ProjectTemplate.csproj is the final file that requires our attention. You can either completely replace its contents with the file **Sample.Project.cs** from Sample.zip and overwrite the handling of different references for different target frameworks, or you can modify it manually to include the correct set of files and assembly references, which will require some knowledge of MSBuild:

- Replace the last `ItemGroup` listing the included source code files with the following two:

```
<ItemGroup>
  <Compile Include="Properties\
  AssemblyInfo.cs" />
  <Compile Include="Sample.cs" />
  <Compile Include="Tests.cs" />
</ItemGroup>
<ItemGroup>
  <None Include="packages.config" />
</ItemGroup>
```

- Add the following assembly reference to the ItemGroup just before these two:

```
<Reference Include="nunit.framework,
Version=2.6.4.14350, Culture=neutral,
PublicKeyToken=96d09a1eb7f44a77,
processorArchitecture=MSIL">

  <HintPath>..\packages\NUnit.2.6.4\lib\
nunit.framework.dll</HintPath>

  <Private>True</Private>
</Reference>
```

Now you are finally ready to create the project template by building Sample.Template project in your solution. If you have done everything correctly, the build should complete without errors. You can find the output file **Sample.Template.zip** in the bin\Debug\ProjectTemplates\CSharp\1033 subfolder of the template project.

For testing, copy it to Visual Studio 2015\Templates\ProjectTemplates subfolder of My Documents and delete the wizard-generated Sample.Project.zip to avoid confusion. Now create a new project from the **Sample** template and notice how both Tests.cs and Sample.cs files are opened in the editor when the project is created. NUnit NuGet package is still restored before the build and the test still succeeds.

Packaging the Template as a Visual Studio Extension

To wrap the project template inside a Visual Studio extension, you will need to add another project to your solution. This time use the **VSIX Project** template (also located in the **Visual C# > Extensibility node**) and name it **Sample.VSIX**. Now double click the **source.extension.vsixmanifest** file in the newly created project to open the editor window. Navigate to the **Assets** page and add Sample.Template as a new project template asset.

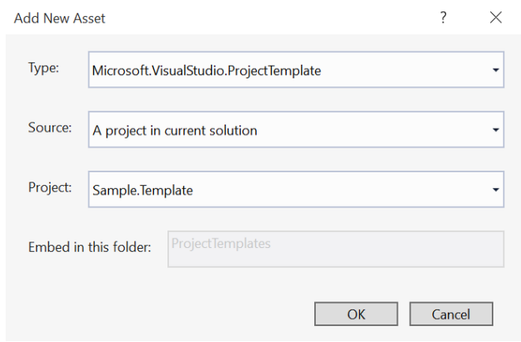


Image 4: Add a new project template asset

The extension should now be ready for testing: just set Sample.VSIX as the startup project and press F5 to start debugging. This will cause Visual Studio to build the extension, run a new instance of Visual Studio using separate "experimental" configuration, and automatically install the extension inside it. When you try to create a new project in this Visual Studio instance, your template will already be available. It should work the same as it did, when you manually installed it, only without affecting your regular Visual Studio configuration.

Before distributing your extension to other developers or publishing it to Visual Studio Gallery, you should fill in the metadata in the common header of the extension manifest editor and on the **Metadata** page:

- **Product Name** will be shown as the title both in the gallery and in the Extension and Updates dialog in Visual Studio.

- **Product ID** must uniquely identify your extension. The default value includes a GUID; therefore, it should satisfy this requirement. You are free to change the value, just keep in mind that the gallery will not allow you to publish the extension, if another extension with the same ID is already published. Do not change this value between versions or you will break the update process for the extension.

- If you want updating of extension to work, you also need to increment the **Version** every time you publish a new version of the extension.

- Edit **Author** and **Description** fields as needed.

- It is a good idea to fill most of the other metadata fields as well, if you plan to put your extension in the gallery, as they will provide more information to potential users.

One last bit of information in the extension manifest that you might want to change, is on the **Install Targets** page. Here you can specify which versions of Visual Studio your extension supports and can be installed into. By default, the value will match the version of Visual Studio you are currently using (14.0 for Visual Studio 2015, 12.0 for Visual Studio 2013), but you can specify a range of supported versions, e.g. [12.0, 14.0] would mean that the extension can be installed in both Visual Studio 2015 and Visual Studio 2013. Of course, you should still test your project template, to make sure it works correctly in all the listed versions. You can also change the product identifier, although the

default choice of Community Edition and higher SKUs should be the right one for most cases.

Conclusion:

The tooling for creating custom Visual Studio project templates is descent, but except for reference documentation in MSDN, there are not many resources about it. The ones that you will find were written for older versions of Visual Studio. The primary goal of this guide is to make you consider project template creation as another tool in your tool belt. If you decide to use it, this guide should be more than enough to get you started ■

Download the entire source code from GitHub at
bit.ly/dncm22-vs-proj-template

About the Author

damir arh

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Microsoft® Most Valuable Professional

BIOLOGICAL MODELING WITH ANGULARJS

Introduction

A few weeks ago I returned back from AngularConnect conference in London. I had the pleasure of speaking about my experience of building a genome viewer for Genome Compiler using both AngularJS and SVG. If you want to watch the session recording it's available on YouTube: https://youtu.be/dPTcdkcrvCY.

In this article I will show you a small use case of how to create a visualization of some biology data using AngularJS. While I won't show you how I created Genome Compiler genome viewer, you will get a taste of how you can create your own visualization. The article will explain how to read a biology formatted file using HTML5 File API, how to use AngularPlasmid library to create a simple visualization and how to build the application using AngularJS.

Disclaimer: this article assumes that you have AngularJS knowledge and won't cover the framework building blocks.

BIOLOGY 101

Before I start to explain how the application works and how I created it, you will need a quick intro to some biology concepts that we will use in the application: the FASTA format and plasmids.

EditorialNote: *You can learn more about AngularJS using our tutorials at* http://www.dotnetcurry.com/tutorials/angularjs . *You can learn more about AngularJS and SVG using Gil's previous article* http://www.dotnetcurry.com/angularjs/1213/create-graphics-using-svg-angularjs

FASTA Format Definitions

In the article we will read FASTA files. The FASTA format is a text-based format that represents a biological sequence. The first line in a FASTA file includes the sequence description. All the other lines in the FASTA file hold the sequence data which is represented in alphabet characters. For example, the following snippet shows a simple FASTA file:

```
>HSBGPG Human gene for bone gla protein (BGP)
GGCAGATTCCCCCTAGACCCGCCCGCACCATGGTCAGGC
ATGCCCCTCCTCATCGCTGGGCACAGCCCAGAGGGTATA
AACAGTGCTGGAGGCTGGCGGGGCAGGCCAGCTGAGTC
CTGAGCAGCAGCCCAGCGCAGCCACCGAGACACCATGAG
.
.
TCCTCTCCAGGCACCCTTCTTTCCTCTTCCCCTTGCCCTT
GCCCTGACCTCCCAGCCCTATGGATGTGGGGTCCCCATC
ATCCCAGCTGCTCCCAAATAAACTCCAGAAG
```

As you can see, the description line (first line) starts with a greater-than (>) symbol which is part of the format. The greater-than symbol might be followed by an identifier and after the identifier there will be a description. Both the identifier and description are optional. In the example we just saw, there is only a description.

Now that we know a little bit about the FASTA format, we can build a small AngularJS service that will parse a FASTA string. In the implementation, I assume that there is only a description and no identifier. So as an exercise, try to implement the whole parser by yourself after reading the article. The following code snippet is the service code:

```
(function () {
 'use strict';
 angular.
  module("biologyDemo").
  factory("fastaParserService",
  fastaParserService);

 function fastaParserService() {
  function readSequence(fastaText) {
   var splittedStrings  = fastaText.
   split('\n'),
    result = {},
    i = 1;
   result.name = splittedStrings[0].
   substr(1, splittedStrings[0].length
   - 1);
```

```
    result.sequence = '';
    for (; i < splittedStrings.length;
    i++) {
     result.sequence +=
     splittedStrings[i];
    }
    return result;
   }
   return {
    readSequence: readSequence
   };
  }
}());
```

In the service code, there is only one function, the **readSequence** function. This function accepts a FASTA string and then parses it and outputs a JavaScript object that includes two properties: a name and the sequence. As you can see, the code is very simple. I use the split function to **split** all the lines in the file. Then, I take the first line, remove the greater-than character and put the output in the returned object name property. The last thing I do is to iterate on all the other lines and concat them to the sequence property.

Plasmids

The second biology concept that you have to know is what are plasmids. A plasmid is a small DNA molecule represented as a circle with some annotations. You can think about it as a graph that represents some DNA of a flu virus or some body cell. For example, the following figure is a plasmid with some annotations:

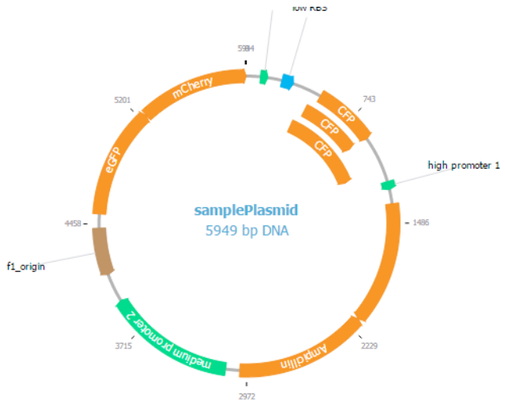


Figure 1: Plasmid

I won't get into further explanations more than what I have already given, since this includes a lot of biology stuff that is not relevant to the article content. You may visit Wikipedia or, search on the internet to know more about plasmids. All you need to understand is that we will probably use a graphics model such as SVG or Canvas to create a plasmid visualization. In the application I will use a library called AngularPlasmid in order to create our FASTA sequence representation. Now that we know a little bit of biology, it is time to return back to HTML5 and JavaScript coding.

Reading FASTA Files Using HTML5 File API

In the application that we are going to create, we will receive a FASTA file as an input, which we will read and parse. In the previous section, we already created the FASTA format parser and in this section you will learn how to read a file using the HTML5 File API.

The HTML5 File API includes a small set of objects that enables you to read files. That doesn't mean that you can just create file readers and read the entire user file system. Browsers restrict the option to read file for security reasons so you will need the user to interact with the application and explicitly allow you to read specific files. You have two points of interactions: drag and drop or file input types. Each of those options will enable you to get a pointer to a file/s to be able to read it.

In the application I'm building, I use a file input type and in the following code snippet you can see the file input type definitions that I use:

```
<input id="fastaFileInput" type="file"
file-reader fasta-content="vm.
fastaFileContent" accept=".fasta" />
```

As you can see in the snippet, in the file input type, I use a **file-reader** attribute which indicates that I have created an AngularJS directive which I'll explain later on. I also use a second attribute that is called **fasta-content** which is going to be a model holder in the application. Once the file is read, the fastaFileContent will hold the **FASTA file content** and we will be able to parse it.

It is not enough to just add a file input type to our HTML in order to read the file. In the directive that we will create, you will use a **FileReader**. The **FileReader** object enables you to read the file. From the file input type you get a file pointer to a file object that includes some file metadata and the file content. Using a **FileReader** you will be able to read the file and then get its content. The **FileReader** object exposes 4 different read functions that you can use to read a file:

1. readAsText – reads the file as string.

2. readAsDataURL – reads the file and returns a URL representing the file's data.

3. readAsBinaryString – reads the file and returns raw binary data representing the file content.

4. readAsArrayBuffer – reads the file and returns an ArrayBuffer that represents the file's content.

The **FileReader** reads the file asynchronously and therefore includes some event handlers such as **onerror**, **onload** or **onloadstart**. Once the file was read, the file's content will be stored in the **FileReader result** property.

Now that we know about the **FileReader** object, it is time to create two new AngularJS objects. The first object will be the **file-reader** directive that will enable us to interact with the file input type. The following code snippet shows you the directive code:

```
(function () {
 'use strict';
 angular.
  module("biologyDemo").
  directive("fileReader", fileReader);
 fileReader.$inject =
 ['fileReaderService'];

 function fileReader(reader) {
  function link(scope, element) {
   element.bind("change", function (evt)
   {
    var file = evt.target.files[0];

    if (file && file.name.indexOf('.
    fasta') > 0) {
     reader.readFile(file).then(function
     (data) {
      scope.fastaContent = data;
     });
    } else {
     alert('Please insert a fasta format
     file!');
    }
   });
  }
  return {
   scope: {
    fastaContent: "="
   },
   restrict: 'A',
   link: link
  };
 }
}());
```

The directive wires an event handler to the file input type **change** event. When we get a new file, we validate it is a FASTA file and then use a service to read the file content and put it in the **fastaContent** scope variable.

The second AngularJS object that we will create is the **fileReaderService** that will be responsible to read the file. The reason I separated this functionality into a directive and a service is the idea of separating DOM interactions and logic code. The following code shows you how the service will look like:

```
(function () {
 'use strict';
 angular.
  module("biologyDemo").
  factory("fileReaderService",
  fileReaderService);
  fileReaderService.$inject = ['$q'];

  function fileReaderService($q) {
   function readFile(file) {
    var deferred = $q.defer(),
    reader = new FileReader();
    reader.onload = function (loaded) {
     deferred.resolve(loaded.target.
     result);
    }
    reader.readAsText(file);
    return deferred.promise;
   }
   return {
    readFile: readFile
  };
 }}());
```

The code of the service is straight forward. Since the **FileReader** API is asynchronous, we will use a promise to wrap the code. The promise is created using the AngularJS **$q** service and is resolved only when the **FileReader** finished reading the file.

Note: *If you don't know what promises are, I suggest reading "*Using Promises in Node.js Applications*" article to get some information about the subject.*

Creating a Plasmid Using AngularPlasmid

After we have read the file and parsed it, we will need a way to visualize our sequence so let's get started to know **AngularPlasmid**. AngularPlasmid is a DNA plasmid visualization component that uses AngularJS and SVG underneath. You can get started with the library on its website: http://angularplasmid.vixis.com/index.php. Once you have download the library and referenced it in your web page, you will need to load its module into AngularJS. The following code shows you how to create the biologyDemo module and load AngularPlasmid module:

```
(function () {
 var app = angular.module('biologyDemo',
 ['angularplasmid']);
}());
```

After you have added AngularPlasmid module, you can start creating visualizations using the directives it includes. The main directive is the **plasmid** element directive which is the drawing space that all the features will be attached to. Once you have the surface, you would probably add a plasmid track. A track is the circular representation that holds other features for the plasmid. You will use the **plasmidtrack** element directive in order to do that. A track can have marking scales which can be created using the **trackscale** element directive. A track can also hold track markers to mark some features in the DNA sequence. You will use the **trackmarker** element directive. For example, the following code snippet shows how to create a simple plasmid created with the AngularPlasmid directives:

```
<plasmid sequencelength='1000'>
 <plasmidtrack radius='50'>
  <tracklabel text='Demo'></tracklabel>
  <trackscale interval='100'
  showlabels='1'></trackscale>
  <trackmarker start='212' end='345'>
  </trackmarker>
  <trackmarker start='530' end='650'>
   <markerlabel text='Ecol'>
   </markerlabel>
  </trackmarker>
  <trackmarker start='677' end='820'>
  </trackmarker>
 </plasmidtrack>
</plasmid>
```

This is just a static representation of a plasmid with scales and markers. The output of running the code will look like the following:

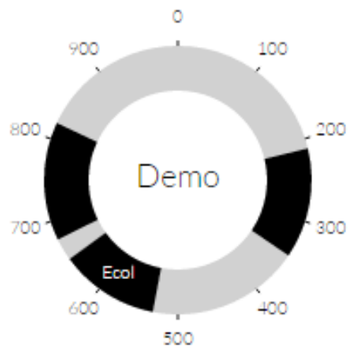


Figure 2: AngularPlasmid Example

Note: *You can explore the entire AngularPlasmid API in the library's web site at:* http://angularplasmid.vixis.com/api.php.

Create the Plasmid Dynamically

One challenge that you will most likely face is how to create the plasmid dynamically according to the sequence we read. The solution for this problem is to generate the entire plasmid elements using regular JavaScript and then use the **$compile** service to compile the elements and give them a scope. The next code snippet shows you a directive that will generate a plasmid when a **data:ready** event is published:

```
(function () {
 'use strict';
 angular.
 module("biologyDemo").
 directive("ngPlasmid",
ngPlasmidDirective);
ngPlasmidDirective.$inject =
['$compile'];

function ngPlasmidDirective($compile) {
 function link(scope, element) {
  scope.$on('data:ready', init);
  function init(e, data) {
   draw(data);
  }

  function draw(data) {
   element.append(createPlasmid(data));
   $compile(element.contents())(scope);
  }

  function createPlasmid(data) {
   var path = document.
   createElement('plasmid');
   path.id = 'p1';
   path.setAttribute('sequencelength',
   data.sequence.length);
   path.setAttribute('sequence', data.
   sequence);
   path.setAttribute('plasmidheight',
   400);
   path.setAttribute('plasmidwidth',
   400);
   path.appendChild(createPlasmidTrack
   ('t1', data));
   return path;
  }

  function addTrackLabel(path, data) {
   var elm = document.
   createElement('tracklabel');
   elm.setAttribute('text', data.name);
   elm.setAttribute('labelclass',
   'tracklabel');
   elm.setAttribute('vadjust', '-12');
   path.appendChild(elm);
  }

  function addTrackScales(path) {
   var elm = document.
   createElement('trackscale');
   elm.setAttribute('interval', '150');
   elm.setAttribute('ticksize',
   'stroke:#000000;stroke-width:1px;');
   elm.setAttribute('style', '100');
   elm.setAttribute('vadjust', '10');
   elm.setAttribute('showlabels', '1');
   elm.setAttribute('direction',
   'out');
   elm.setAttribute('labelclass',
   'trackscale-label');
   path.appendChild(elm);
  }

  function createPlasmidTrack(id, data)
  {
   var elm = document.
   createElement('plasmidtrack');
   elm.setAttribute('id', id);
   elm.setAttribute('radius', '133');
   elm.setAttribute('width', '3');
   elm.setAttribute('fill', '#d5d6db');

   addTrackLabel(elm, data);
   addTrackScales(elm);
   return elm;
  }
 }

 return {
  restrict: 'E',
  replace: true,
  scope: {   },
  link: link
 };
}
}());
```

The interesting function in the directive is the **draw()** function. In the **draw** function, I append the plasmid I generate using the given data to the directive's element. Then, I compile the element content and attach to it the directive scope. That will do the magic and create the data binding for us. Now that we know how to generate a plasmid representation, it is time to combine all the things that we learned so far and create a whole application based around it.

The Full Example - Biology Modeling with AngularJS

In the previous sections, you already saw some of the components that we will use to create the application. Now all we need to do is to join all the parts together. We will start with the main web page:

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>Biology with Angular</title>
 <link href="styles/main.css"
 rel="stylesheet"/>
</head>
```

```
<body>
 <div ng-app="biologyDemo">
  <section class="body-content">
   <ng-biology-demo></ng-biology-demo>
  </section>
 </div>

 <script src="app/vendor/angular/
 angular.min.js"></script>

 <script src="app/vendor/angularplasmid/
 angularplasmid.min.js"></script>
 <script src="app/app.js"></script>
 <script src="app/biology/services/
 fastaParserService.js"></script>
 <script src="app/biology/services/
 restrictionSiteService.js"></script>
 <script src="app/common/services/
 fileReaderService.js"></script>
 <script src="app/common/directives/
 fileReaderDirective.js"></script>
 <script src="app/common/controllers/
 demoController.js"></script>
 <script src="app/common/directives/
 ngDemoDirective.js"></script>
 <script src="app/common/directives/
 ngPlasmidDirective.js"></script>
</body>
</html>
```

Nothing interesting happens here and the only thing to notice is the usage of **ng-biology-demo** directive. Here is the directive code:

```
(function () {
 'use strict';
 angular.
 module("biologyDemo").
 directive("ngBiologyDemo",
 ngBiologyDemo);
 ngBiologyDemo.$inject = [];
 function ngBiologyDemo() {
  return {
   restrict: 'E',
   templateUrl: 'app/common/templates/
   demoTemplate.html',
   controller: 'demoController',
   controllerAs: 'vm'
  };
 }
}());
```

Again, the directive itself isn't interesting but it has a controller called demoController and a template that we load. The template code looks like this:

```
<div>
 <div>
  <label for="fastaFileInput">Select a
  Fasta File: </label>
  <input id="fastaFileInput" type="file"
  file-reader fasta-content="vm.
  fastaFileContent" accept=".fasta" />
 </div>
 <div>
  <ng-plasmid></ng-plasmid>
 </div>
</div>
```

And the controller code looks like this:

```
(function () {
 'use strict';
 angular.
  module("biologyDemo").
  controller("demoController",
  demoController);
 demoController.$inject = ['$scope',
 'fastaParserService'];

 function demoController($scope,
 parser) {
  var vm = this;
  function init() {
   vm.fastaFileContent = '';
   $scope.$watch(function () {
    return vm.fastaFileContent;
    }, function(changed) {
     if (!changed) {
      return;
     }
     $scope.$broadcast('data:ready',
     parser.readSequence(vm.
     fastaFileContent));
    });
   }
   init();
  }
}());
```

In the controller, we watch the **fastaFileContent** property. The property will only change when the file is read by the **file-reader** directive **change** function. Once the content changes, we parse it and raise the **data:ready** event to notify the plasmid directive that its data is ready to use. Then, the plasmid directive will generate our plasmid.
Since the generated plasmid won't have markers, I decided to add a service that will find the occurrences of some sub sequence and we will use it in the plasmid directive to add a marker. The following code snippet shows you how the service looks like:

```
(function () {
 'use strict';
 angular.
  module("biologyDemo").
  factory("bioService", bioService);
 function bioService() {
  function findMatches(sequence, pattern)
  {
   var result = [],
    indexFound = sequence.
    indexOf(pattern);
   while (indexFound > -1) {
    result.push({
     start: indexFound,
     end: indexFound + pattern.length
    });
    indexFound = sequence.
    indexOf(pattern, indexFound + 1);
   }
   return result;
  }
  return {
   findMatches: findMatches
  };
 }
}());
```

The service exposes a function called **findMatches**. The function is responsible to find matches in the sequence and return them as an array of start and end points. In the plasmid directive, you will add a new function called **createTrackMarkers** and for simplicity we will use a constant pattern with it:

```
function createTrackMarkers(elm, data) {
 var pattern = 'CTGCAG',
  matches = bio.findMatches(data.
  sequence, pattern),
   i = 0,
   marker,
   markerLabel;
 for (; i < matches.length; i++ ) {
  marker = document.
  createElement('trackmarker');
  marker.setAttribute('start',
  matches[i].start);
  marker.setAttribute('end',
  matches[i].end);
  marker.setAttribute('class', 'track-
  marker');
  marker.setAttribute('wadjust', '10');
  marker.setAttribute('vadjust', '-4');
  marker.setAttribute('markerstyle',
  'stroke: yellow;fill: blue;');
  markerLabel = document.
  createElement('markerlabel');
  markerLabel.setAttribute('class',
  'markerlabel-inside');
  markerLabel.setAttribute('text',
  pattern);
  markerLabel.setAttribute('type',
  'path');
  markerLabel.setAttribute('vadjust',
  '15');
  marker.appendChild(markerLabel);
  elm.appendChild(marker);
 }
}
```

Note: *In real world applications, the pattern will be dynamic and will probably arrive from a user or from another service.*

In the **createPlasmidTrack**, add a call to the **createTrackMarkers** function before you return the **plasmidtrack** element and of course add an injection to the **bioService** in the plasmid directive. Now you can run the demo. If you read a FASTA file such as the file content showed in "FASTA Format Definitions" section, you will get the following output:

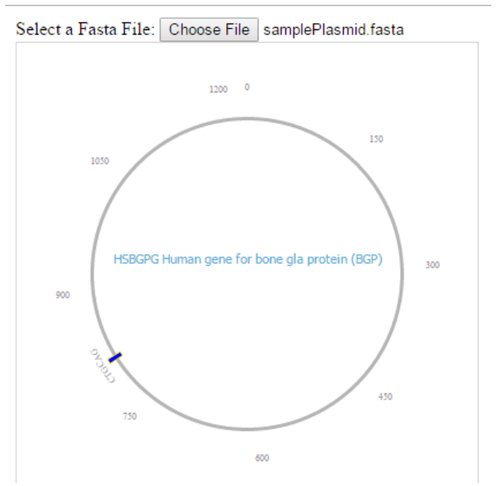


Figure 3: The Application Running

Summary

This article is probably not your ordinary coding article. We explored a few concepts in AngularJS and JavaScript but also some biology concepts as well. My hope is that you will take this small application I built and try to understand how it is structured and implemented.

All in all, the creation of biology models in JavaScript is very challenging. My hope is that you got some hints of how to create interesting things that you can play with and maybe use it as a starting point to build your own biology oriented application ■

Download the entire source code from GitHub at bit.ly/dncm22-angularjs-plasmid

About the Author

gil fink

Microsoft® Most Valuable Professional

Gil Fink is a web development expert, ASP.NET/IIS Microsoft MVP and the founder of sparXys. He is currently consulting for various enterprises and companies, where he helps to develop web and RIA-based solutions. He conducts lectures and workshops for individuals and enterprises who want to specialize in infrastructure and web development. He is also co-author of several Microsoft Official Courses (MOCs) and training kits, co-author of "Pro Single Page Application Development" book (Apress), the founder of Front-End.IL Meetup and co-organizer of GDG Rashlatz Meetup. You can get more information about Gil in his website http://www.gilfink.net.

Process Customization in Visual Studio Team Services

Microsoft recently made an announcement about a Visual Studio Team Services feature that was due for quite some time. Microsoft now allows limited customization of Process in Visual Studio Team Services. Considering that VS Team Service is a multi-tenant service offered in the cloud, this is a big step towards allowing total customization of the Process.

*If you are wondering **'what is Visual Studio Team Services'**, then let me clarify that it is the same set of services which were available as Visual Studio Online (VSO), renamed as Visual Studio Team Services or VS Team Services. In a nutshell, it refers to TFS in the cloud.*

So far, the only process templates that were allowed to use on VS Team Service were the ones available out of box - CMMI, Agile and SCRUM. Although they catered to majority of common needs of customers, there were quite a few customers who wanted to have their own definitions of work items, their own work item types and workflows for existing work items. Using a cloud based service like VS Team Service reduces the cost of using the product because we share the infrastructure with other customers. Sharing the infrastructure is good from a cost perspective, but not so good for customization and that's why Microsoft did not offer customization of VS Team Services earlier. Microsoft knew that not providing the ability to customize, was going to be one of the road-blocks for adoption of VS Team Services. For the last few months, efforts were on to provide some customizability to the processes of VS Team Services and now we are seeing the result of that.

To reach the interface of process customization on the VS Team Service, the route is from 'Overview' page of your account > to Account management (Control Panel) page > to View collection administrator's page > to Process tab on that page. When we reach that page, we see the three out of box process templates – Agile, CMMI and SCRUM.

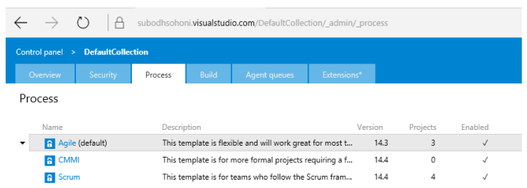


Figure 1: Existing Processes in VS Team Services

Until now we could export each of these services if needed. Now we get many more options when we click the triangular black icon on the left of any process template name.

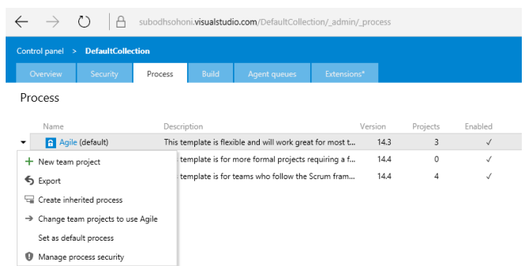


Figure 2: Inherit a Process

Our journey for customization starts by creating the inherited process which we will change as needed. Let us create a process named 'SSGS Agile Process' inheriting it from 'Agile'. One thing that you will notice immediately is that you cannot create a new work item type. Let us take the example of our TicketM software. For our TicketM software, we would like to create a Ticket work item type but at this moment there does not seem to be a way to add a new work item type. This is a major setback to what we wanted to do as part of customization of process. OK, so we are constrained to edit an existing work item type only. Let's hope that within the next couple of months, Microsoft will provide the feature where we can add a new work item type.

Going forward, what we would like to do in the existing work item type definition is the following:

- Create a new field named Customer for a User Story. It should be just a string field with allowed values that for the sake of assumption, are 13 customer names that we have. We would like to assign a default value to it.
- Change the workflow of the User Story so that we have a state for 'Blocked' or 'On Hold' with some related transitions.
- We also expect that we will be able to create a dependency such that if the state is 'Blocked' it should be compulsorily assigned to a user in a specific group of the team project.

With these limited goals let us set out on our journey.

On the newly created process template, we can see a tab for Work Item Types and Fields. We do not see the tab for workflow which is something we wanted. Unless it is hidden somewhere under the Work Item Types, we will have to wait for that too. Out of existing tabs, the Fields tab only listed the various fields for view, which are present in the bucket of fields. It listed those with the work items that use each field. This is good. We would like to see something like this for TFS too.

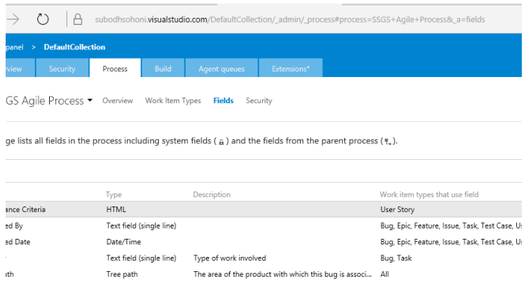


Figure 3: Fields List

The tab for work item types is more suitable for making modifications to work item type definitions. It lists the work item types in the inherited process template and for each of those work item type, it lists the sections for Overview, Layout and Fields. The overview node shows the fields that are not editable. These fields are the name, description and the color with which it shows the work item in the query results.

Although these fields are read-only, the screen mentions that they are read-only 'at this time' which gives an indication and a hope that they *may* become editable in future.

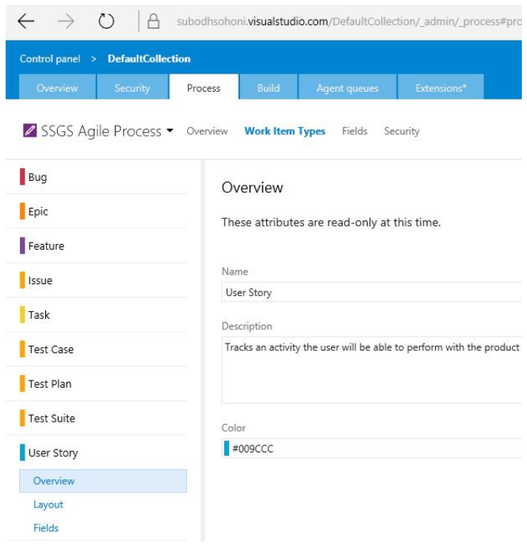


Figure 4: Work Item Overview

The Fields node is the one that shows the categorized list of fields in the selected work item type. Here we will select User Story, and add a new field to the list. We can select a field in the list and edit its definition.

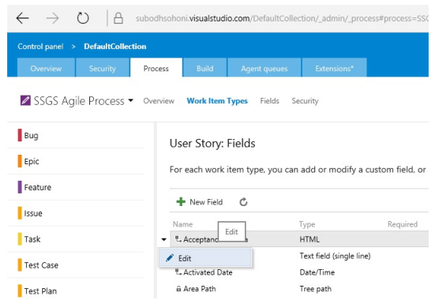


Figure 5: Edit a Field

What the editor shows is the field definition attributes like Name, Type and Description. These attributes are not editable while editing a field, but are editable when the field is being created. Which means that we can make no mistakes while creating the field definition. There is no chance for correction in case of any error once the field is saved.

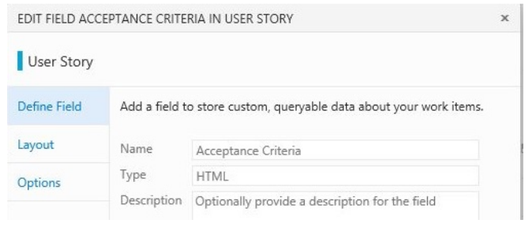


Figure 6: Field Definition

In the Layout node, it allows us to set whether that field can be viewable on the form of the work item of that type. It allows us to edit the Label that appears beside the field, and also allows us to place the field in any of the existing groups on the form of the work item type. We can also create new groups that will make logical grouping of new fields

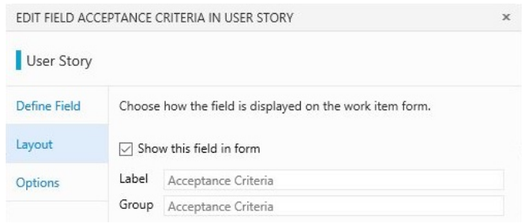


Figure 7: Field Layout

And finally the options node is where we can set some rules on the field. The first one is whether the field is a mandatory field or not. We can also set the default or initial value of the field. This also means that we cannot set any of the more complex rules that are available in customization of on-premise TFS.

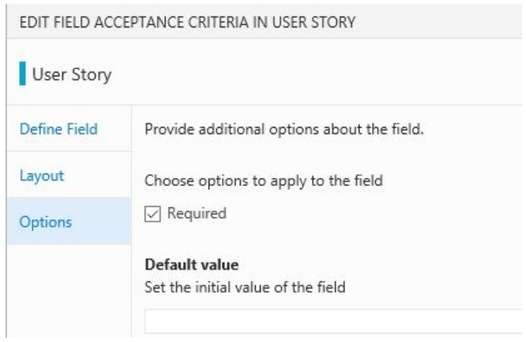


Figure 8: Field Options

When we save these details, we can view the fields list again with the changes appearing in the respective columns.

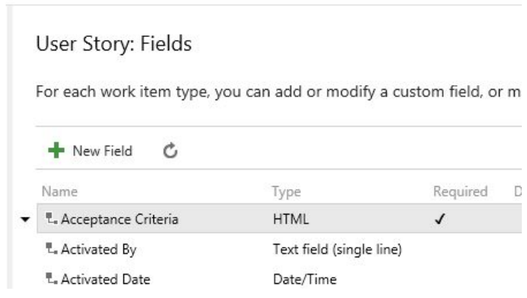


Figure 9: Field List After Field Edit and New field

Let us now add a new field named Customer as desired by clicking on the button of New Field. We will set it to a text field type. We come to realize that here we have no way to provide 'Allowed Values' which are our customer names. This is something that we will need in the earliest updates of this feature.

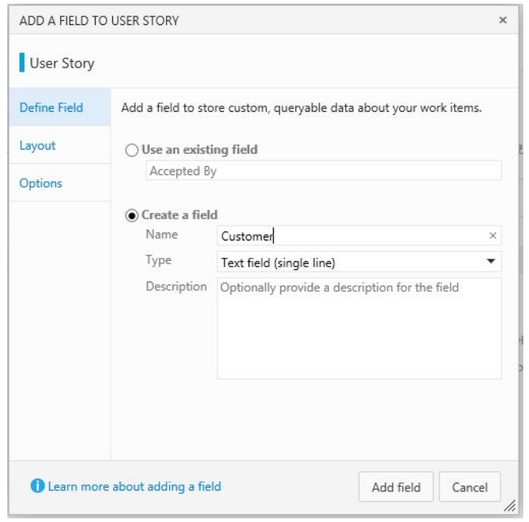


Figure 10: New Field

After giving the Classification as the group in which to include this field and the option of Required, we will now save the field. It now became part of the process 'SSGS Agile Process'

When we create the new team project with the newly created process template and create a new User Story in it, we can see the Customer field in the classification section in it.

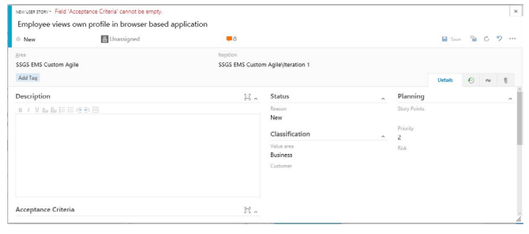


Figure 11: New Field View

One very nice feature that in my opinion is awesome, is that we can apply this process template to the existing team projects too.

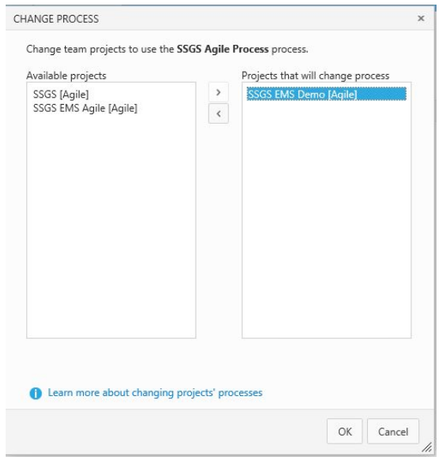


Figure 12: Apply Process to Existing Team Project

Summary

Microsoft has taken a baby step towards providing the customization of the Process in Visual Studio Team Services. Although I expected more, right now only possibility is to do customization of fields in work items. Even that is quite limited. I do expect much more out of this feature in the near future. Since it is being done on a multi-tenant services, I do commend Microsoft for providing at the least some beginning to this much awaited feature ■

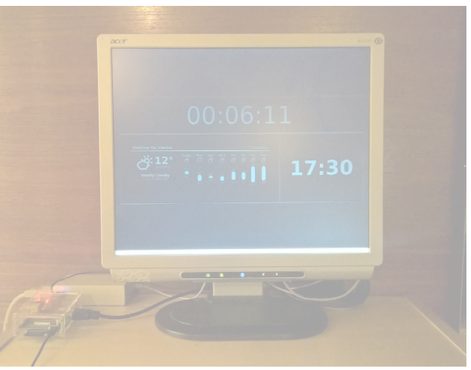
About the Authors

Subodh Sohoni, Team System MVP, is an MCTS – Microsoft Team Foundation Server – Configuration and Development and also is a Microsoft Certified Trainer(MCT) since 2004. Subodh has his own company and conducts a lot of corporate trainings. He is an M.Tech. in Aircraft Production from IIT Madras. He has over 20 years of experience working in sectors like Production, Marketing, Software development and now Software Training. Follow him on twitter @subodhsohoni

HomePi

A Windows 10 IoT app

(Part 1)



I use public transport to commute to office every day and use the official transport mobile app to check my bus timings. Instead of checking the app every day, I came up with an idea of using a Raspberry Pi and a free monitor to display a countdown for my next bus.

When Windows 10 IoT was released, I decided to make the following changes to this project

1. Move the project to Raspberry Pi 2

2. Develop a new Universal Windows Project (UWP) and add a couple of additional features like Weather, Music Player, Power Controls, Email Notifications etc.

3. Replace the monitor with a LCD display to cut down power

In this **series of articles**, we will see how I converted the above setup into a better version with a Windows 10 IoT app. We will call it **Home Pi**. This part of the article will display the weather information. We will add more features as we progress through this series.

Here is how the new setup looks like:

Hardware & Requirements

Following is the list of parts required for our project

1. Raspberry Pi 2 Running Windows 10 IoT

2. 3.2" TFT LCD Module 320*240 Touch Screen Display for Raspberry Pi

This LCD display is powered by ILI9341 display controller and TSC2046 touch controller. Currently there is no official support for LCD displays in Windows 10 IoT so we will be writing custom display driver and touch processor for our project so any LCD board which is powered by above controllers will work with our code. Here is another model https://www.conrad.at/de/raspberry-pi-display-modul-schwarz-rb-tft32-v2-raspberry-pi-a-b-b-raspberry-pi-1380381.html

I highly recommend reading the following articles before starting the project to know more about how the Touch and display controllers are implemented.

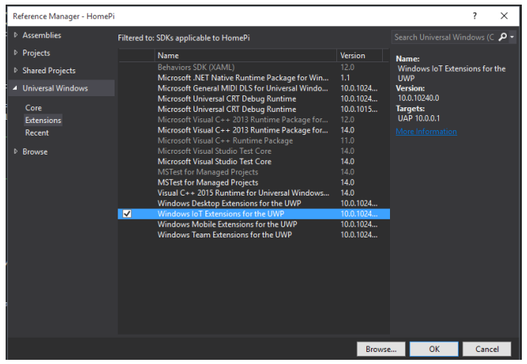
1. Building a Touch-enabled Interface for Windows IoT

2. Raspberry Pi 2, 480*320 TFT LCD Displays Windows 10, SPI

Windows 10 IoT Project

Open Visual Studio 2015 Community Edition. Start by creating a new Universal Windows Project by doing File -> New -> Project -> Blank App (Universal Windows) under Visual C# templates.

Right Click the newly created Project in Solution Explorer -> Add -> Reference. Under **Universal Windows** Select **Extensions** and select **Windows IoT Extension for the UWP**. Click OK to add the extension to the project.



Right click **Package.appxmanifest** file and click **View Code**. Add the following capabilities under Capabilities tag

```
<uap:Capability Name="documentsLibrary"
/>
```

This will give our app enough permission to store the Touch Screen calibration data in Documents Library folder.

Register for a free developer API on http://www.worldweatheronline.com/api/. We will need the API key to fetch weather information.

Project Files

Please download the source code of this article to understand the important files of this project. Comments wherever needed have been added in the source code itself.

Tsc2046.cs

This file under Class folder has methods to save calibration data to Documents Library and Load Calibration data when the app starts. This way our app will receive proper touch inputs while using touch features.

LocalWeather.cs

This class file is used for processing weather information received from World Weather Online API Service.

Visit http://www.worldweatheronline.com/api/docs/local-city-town-weather-api.aspx for more information about the API.

wwoConditionCodes.xml

This xml file contains the list of Weather Codes returned from World Weather Online and their corresponding Weather Description and Icons. We will use this information to load Weather Icon from local storage instead of fetching it every time.

ILI9341.cs

This file acts as the display driver and has helper methods for displaying images, moving cursor to a specific point, writing text to screen etc.

MainPage.xaml.cs

This page, when loaded, checks for calibration data file. If the data file is present, the program proceeds to display the current weather information. If calibration data is not present, one-time calibration setup is run and 5 calibration points are stored for future use. Weather is updated every 1 hour with the use of a timer.

***Note**: Make sure to update your API key and location information in GetWeather() method.*

Init()

This method initializes the LCD Display, Touch sensor and the Weather timer to run every 1 hour. It also tries to load the calibration file from Documents folder and if not found, goes to calibration mode by calling `CalibrateTouch()` method. Once the calibration is done, the screen is painted with the colour White.

CalibrateTouch()

This method is used to calibrate the touch screen using 5 points. Touch points are displayed one after another after Touch input for each point is received. These values are stored in Documents Library using `SaveCalData()` method.

GetWeather()

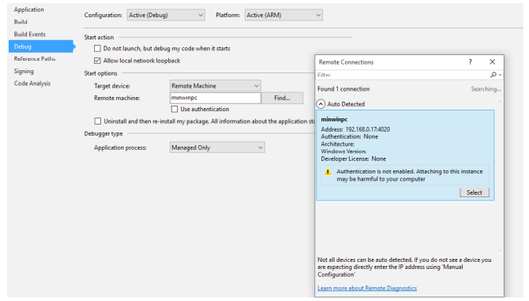
GetWeather method starts by displaying an Image which will be used as a background and menu.

In a future article we will receive Touch input and match the x and y coordinates with the x and y coordinates of the menu icons (Bus Countdown, Music Player, Weather, Power).

The text "Checking..." is displayed while the weather details are fetched using `getJsonStream()` method by passing the local weather url with the API key and location info. Once the weather info is available, this method paints a white rectangle to clear the status text. `wwoConditionCodes.xml` is used to display the appropriate weather icon based on the code. Day & Night icons are chosen based on time. Weather description is displayed based on the length of the string.

Remote Deployment

Right click project and select Properties. In the resulting window, under Debug tab select Target Device as Remote Machine and select your Raspberry Pi.



Press F5 to deploy and debug the project. If this is the first time you are running the project, you will see the Calibrating setup screen. Simply tap the dots displayed in the screen to calibrate the screen. Once the calibration is successful, you will see your weather information.

In the next article, we will add a countdown for the next bus, a simpler version of our Sound Cloud Music player and some power controls. Stay tuned!■

Download the entire source code from GitHub at bit.ly/dncm22-windows10-homepi

About the Author

Shoban Kumar is an ex-Microsoft MVP in SharePoint who currently works as a SharePoint Consultant. You can read more about his projects at http://shobankumar.com . You can also follow him in Twitter @shobankr

OVERVIEW OF ASP.NET 5

ASP.NET 5 is a significant redesign of ASP.NET. It converges ASP.NET MVC and Web API together as MVC 6. ASP.NET 5 applications can now be installed and run on Linux and Mac OS X.

ASP.NET 5.0

ASP.NET 5: CONCEPTUAL OVERVIEW

ASP.NET 5 is a significant redesign of ASP.NET. Unlike other versions of the technology, this version is not an enhancement to the previous version. It is written from scratch to make the technology better and lighter.

You may have a question as to what made Microsoft take such a bold step and rethink about everything that they built over the years and start afresh. The reason is, the new platform competes with other web platforms like Node.js and Ruby and this competition makes it necessary to make the platform lighter and crop existing platform as they stand today.

In this article we will take a look at the architecture of the platform and will understand some of the most essential pieces.

ASP.NET 5 - WHAT'S IN IT?

ASP.NET 5 is created with modularity and performance in mind and to make the technology available on as many platforms as possible. Dependency on System.Web.dll has been removed for this purpose. The System.Web.dll works only on Windows platform taking advantage of certain features that only Windows can offer, so there was no way it could be made cross platform. The removal of System.Web.dll has made ASP.NET Web Forms distant from ASP.NET 5.

Editorial Note: *Microsoft continues to invest in Web Forms including adding support for Roslyn. Some additional features added to Web Forms is HTTP 2 support and async model binding. To learn more, please refer to a fantastic article by Rion Williams on What's New in ASP.NET Web Forms in .NET 4.6* http://www.dotnetcurry.com/aspnet/1127/aspnet-webforms-new-features

In ASP.NET 4.5, ASP.NET MVC and Web API used to work in a similar fashion, but they were built on completely different set of class hierarchies under the covers. ASP.NET 5 converges these two components together and provides a unified way of creating both views and APIs. Views and objects are just two different return types of the base Controller class. As we have the same controller handling both types of requests, we can use the same set of action filters and same routing for both of the response types.

Supported .NET Frameworks

In case you are not aware, Microsoft created a new light weight and cross platform version of the .NET framework. It is called .NET core. The .NET core includes a subset of class libraries from the full .NET framework and is designed to work on multiple platforms. It has a corresponding runtime called CoreCLR. This .NET core does not contain libraries like System.Drawing, System.Windows and other libraries that are dependent on the Windows platform to achieve their functionality.

ASP.NET 5 is designed to work on both the full .NET version, as well as on the .NET core version. The code that is targeted to run on .NET core can be executed on any platform. ASP.NET 5 runs on Roslyn, so the builds are completely dynamic. That means in order to see the updated result after making changes to any C# file in the project, we don't need to build the project. We can simply refresh the browser to see the updated result.

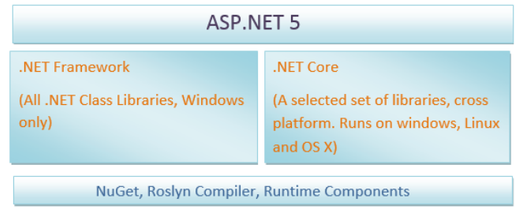


Figure 1: Core components of ASP.NET 5

How does an ASP.NET 5 application work?

When an ASP.NET 5 application starts, it runs only the essential components that are needed to start the server and serve simple pages. Unlike previous versions of ASP.NET, it doesn't start a set of features

that the application may or may not use. If we need a feature, we need to invoke the respective middleware. MVC, Entity Framework, Identity, Caching, Logging and any services that we need to use in lifetime of the application have to be loaded as middleware services.

ASP.NET 5 contains a built-in IoC container. It can be used to configure dependencies for the application. It is possible to replace it with an IoC container of your choice.

The Solution explorer window of a new ASP.NET 5 project created on Visual Studio 2015 looks like figure 2:

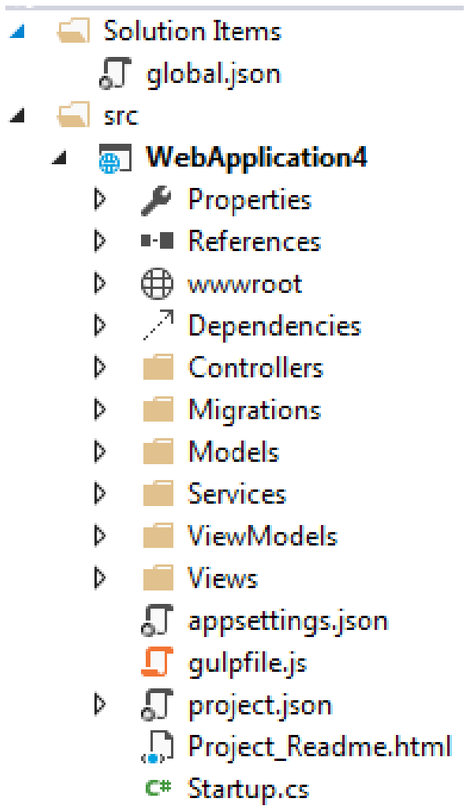


Figure 2: ASP.NET 5 Project structure

It has the following things at its core:

- project.json: This file replaces Web.config and package.config files from previous versions. The JSON schema of this file is stored at schemastore.org. Following are some of the important configurations provided in this file:

  o version: version of ASP.NET 5 to use

  o compilationOptions: These are passed to Roslyn for compiling the code. It specifies the version of C# to target, whether to allow unsafe code and a few other options

  o commands: These are the commands that can be executed on a PowerShell command prompt on Windows and on terminals of OSX/Linux if DNX (Dotnet Execution Environment) is installed. In the project.json file created by the template on Visual Studio, you will find the command web assigned with *Microsoft.AspNet.Server.Kestrel*. Kestrel is a cross platform web server to run ASP.NET applications from command line and it helps us in running ASP.NET 5 apps on Linux and OSX

  o frameworks: It is the list of frameworks on which the application can run. You can specify multiple versions and use one of them depending on the platform on which the application runs

  o dependencies: It is the list of NuGet packages with their versions and the list of class libraries that the project needs to run. If you are using Visual Studio, the dependencies are loaded as soon as you add a dependency and save the file. The list of installed dependencies can be found under references section of the application. If you edited this section using other editors on Windows or on other platforms, then you need to run the command "dnu restore" to get the dependencies installed. DNU (Dotnet Execution Environment Utility) is the tool that helps in dealing with packages

  o scripts: These are the tasks to be executed when the application is built or published. It is used to install front-end dependencies using NPM, bower or other package managers, run tasks using grunt or gulp in order to make the things ready when the application runs

- Startup.cs: It is the starting point of the ASP.NET 5 application. If you are familiar with Katana project, this file is inspired from the same project. Name of this file is by convention. It has a static void main method and as you would have already guessed, this method is the starting point of the application. The methods in this class are used to load middleware services, configure abstract and concrete types for IoC container, to define routes and to configure any service that is loaded as a middleware. The methods *Configure* and *ConfigureServices* are defined

- global.json: This file is present at root of the application folder and it contains solution level settings. By default, it contains names of the projects and SDK version. These settings are used for all projects under the solution.

- appsettings.json: This file is used to store application settings. It may include things like database connection string, mail server to use while sending mails, folder path where uploaded files have to be saved and any such settings that are used by the application. Name of this file is not by convention. It is read in the Startup.cs file. We can store environment specific application settings in individual files and load one of them according to the current environment.

- gulpfile.js: This file defines a set of gulp tasks to copy, combine, minify, uglify and clean the front-end files and dependencies. The tasks can be executed via command line or, can be configured to run during build or publish in the project.json file.

- package.json, bower.json: By default, these files are not visible in the solution explorer, we need to select the view all files option to see these files. The package.json file contains list of NPM dependencies and the bower.json file contains list of bower dependencies to be used by the project. The installed dependencies can be seen under the *dependencies* section of the solution explorer

- wwwroot: This is the folder that would contain JavaScript, CSS, images and other static assets required by the application. Tasks defined in gulpfile.js copy the final static files in this folder.

ASP.NET 5 Request LifeCycle

Now that we know the different pieces of an ASP.NET 5 application, let's now take a sneak peek of what happens when a URL of an ASP.NET 5 application is invoked. Figure 3 shows what goes on after the request reaches the web server:

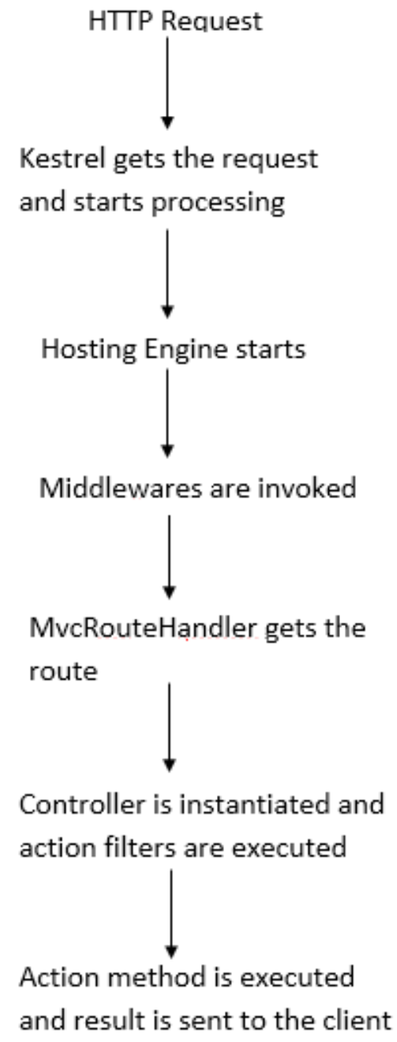


Figure 3: ASP.NET Request Life Cucle

Conclusion

ASP.NET 5 is created from scratch to make the platform better and to embrace the Node.js ecosystem inside Microsoft's tools. This change makes the developers working on ASP.NET stay abreast with the latest open source tools and it opens up the opportunities to write and run ASP.NET code on multiple platforms. Yeoman has a few generators to scaffold ASP.NET 5 applications, they can be used on non-windows platforms to quickly start writing an application. I hope this article gives you a good start to ASP.NET 5. We will have more articles on this topic in future ■

About the Author

ravi kiran

Microsoft® MVP Most Valuable Professional

Rabi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

THANK YOU

FOR THE 22nd EDITION

@damirarh

@craigber

@shobankr

@gilfink

@subodhsohoni

@rionmonster

@sravi_kiran

@suprotimagarwal

@saffronstroke

JOIN OUR TEAM