Developer's Guide to Support as a Platform

Platform Overview

Blizzard Customer Support uses Atlas to continue delivering our usual Epic service to players, even while they're having issues. Using Atlas, our Game Masters solve issues as fast as they can to get players back into the game.

Service Technologies built the Atlas Platform API to provide a simple, standardized interface for our ever-expanding catalog of Blizzard and partner products. This API gives product teams the ability to read and manipulate player data without the usual complexities between Support tools and in-game implementations.

Guide Summary

This guide will show you how to use standard Service Tech libraries to develop .NET Core applications that are compliant with the Platform APIs. Using these libraries makes it easier to keep up-to-date with API changes, and should minimize the amount of boilerplate code needed.

Creating a Project

All projects start their life as an ASP.NET Core Web Application in Visual Studio.

1. Select **Create a new project** from the Visual Studio start menu
   Note: You can also select **File** > **New** > **Project** or click the **New Project** toolbar button
2. Select the **ASP.NET Core Web Application** project template and click **Next**
3. Select the **API** application type and click **Create**
    a. (Optional) Check **Configure for HTTPS** and **Enable Docker Support** if your project needs them
4. After creation, make sure your project's **Target Framework** is set to **.NET Core 3.1**

Table of Contents

Importing Atlas.Platform.Api.Controllers

After you create a project, you need to import the shortcut libraries. Add a *nuget.config* file at the solution level with the following repositories in addition to your team-specific repositories.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <add key="Blizzard Shared" value="https://nuget.battle.net/api/nuget/nuget-blizzard" />
        <add key="Nuget" value="https://nuget.battle.net/api/nuget/org-nuget" />
    </packageSources>
</configuration>
```

Adding these repositories provides access to the main nuget repository and the Blizzard shared repository, where the Support Platform packages reside. After adding the package, the default support controller will give you access to the following standard behavior endpoints.

Method	Endpoint	Description
GET	/api/atlasSupport/collection	Gets collection data from the platform adapter
GET	/api/atlasSupport/item	Gets items from the platform adapter
POST	/api/atlasSupport/command	Processes commands
POST	/api/atlasSupport/customerResponse	Used to send in-game customer responses from CS actions and interactions

Implementing the *game* dataType

Platform adapters need to return one or modules, representing individual data items to expose to Support. The *game* data type is required for all adapters as its the main starting point for interactions with that type. Atlas will always offer to show the game node for every game on a player's account.

Creating the Item Module Factory

To create a new item module factory, add a new class derived from Atlas.Platform.Api.Controller. ItemModuleFactory to the GameItemModuleFactory program. There are several constructor parameters needed by the base class. This implementation provides most of these, but the collectionFactories needs to come from the constructor.

Note: There are two versions of the Item Module Factory. One takes a generic type parameter, the other does not. In most cases, GameItemModuleFactory should use the non-generic class. For more information on the generic version, click here.

Your game item module should look similar to this:

```
public class GameItemModuleFactory : ItemModuleFactory
    {
        public GameItemModuleFactory(IEnumerable<ICollectionModuleFactory>
collectionFactories)
                : base("game",
                "",
                true,
                collectionFactories)
        {
        }
    }
```

Item Module Parameter Notes

- The game module data type must be *game*. Other item data types are up to you, but they must be unique across products. There are also reserved names, like *game*, that you can find in our platform documentation.
- If you want to include a font for button icons, you can use *tagFont*. Standard icons are documented on our Confluence, but you can also work with Service Tech Product Design to implement a custom icon font.
- To record access to this data type through Telemetry, make sure you mark *requiresAudit* as true.

Registering the Module Factory with the Application

After creation, you need to register the class with the service container so the controller can instantiate a factory instance for responses to Atlas requests.

To do so, use the ConfigureServices method as shown below.

Note: You should use the IItemModuleFactory interface to register the factory. This interface also includes temporary stub implementations to prevent construction errors.

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();

                        // Stubs
                        services.AddInstance<ICollectionModuleFactory[]>(new ICollectionFactory[0]);
                        services.AddInstance<ICommandHandler[]>(new ICommandHandler[0]);


                        // Module Factory Registrations
            services.AddScoped<IItemModuleFactory, GameItemModuleFactory>();
        }
```

Adding Collections

Collections let us display, and allow control to, certain aspects of the game to our GMs. Each collection represents one type of data to expose for display and manipulation. A group of collections is built using Tags.

For the following tutorial, our fictional product contains a collection of characters, a collection of account-level toys, and a collection of account-level achievements. The character collection will use a stand-alone *Characters* tag while the other collections will use a tag named *Unlocks*.

Adding Tags

To start, you need to add the tags to the *game* Item Module by updated the GameItemModuleFactory with the following:

Note: Each item can have a set of tags to group its collections.

GameItemModuleFactory.cs

```
public GameItemModuleFactory(IEnumerable<ICollectionModuleFactory> collectionFactories)
  : base("game",
        "",
        true,
        collectionFactories)
  {
    AddTag("characters", "characters", "Characters");
    AddTag("unlocks", "lock", "Unlocks");
  }
```

The AddTag method takes a name, an icon id, and a caption. The provided icon is used in Atlas for the group and caption—in English—is the tooltip. If you specified a tagFont, then the icon will come from the icon font, otherwise it will come from the standard Atlas collection.

After you create tags, it's time to add the basic collection support. Your first collection provides a list of characters. For this example, assume there is a pre-existing back-end service to provide these characters.

```
public class Character
    {
        public ulong ServerId { get; set; }
        public ulong CharacterId { get; set; }
        public uint ClassId { get; set; }
        public string Name { get; set; }
        public uint RaceId { get; set; }
        public uint Level { gets; set; }
        public DateTime CreatedOn { get; set; }
        public bool IsDeleted { gets; set; }
        public bool DeletedOn { get; set; }
        public bool CanUndelete { get; set; }
    }
```

This class is a simple data transfer object (DTO) and represents what you may want to display on the character.

Creating a Collection Module

Add a new class called CharacterCollectionModuleFactory, derived from CollectionModuleFactory, and use Character as the TypeParameter. The list of paramaters for this class are shown to the right.

We have a service that provides the list of characters, so we will forward that data using the GetRawData call. This call has several parameters, shown in a table to the right.

The class should look something like this when you're done:

```
    public class CharacterCollectionModuleFactory :
CollectionModuleFactory<Character>
    {
        private readonly ICharacterService _characterService;

        public CharacterCollectionModuleFactory(ICharacterService
characterService)
            : base("characters",
                "Characters",
                "No Characters",
                true)
        {
           _characterService = characterService;
        }

        protected override Task<List<Character>?> GetRawDataAsync(int?
titleId, string key, string? issueId)
        {
                        return _characterService.GetCharacters(key); //
The key is defined by the collection and can be parsed
                                                                // by the
platform adapter to whatever format is needed
                                                                // for the call
        }
    }
```

Atlas.Platform.Api.Controllers has nullable references turned on, so you will see *List<T>?* to represent it's possible to return *null* for a data type. In general, you should only return null from *GetRawDataAsync* if you don't support this data type.

Collection Module Parameters

Name	Description
dataType	The unique data type to identify a collection. **Example**: *characters*
name	The display name shown in the tab of the collection's parent item.
emptyCollectionPrompt	The text shown to a GM when a collection is empty, used to distinguish between empty and errored collections.
requiresAudit	Used to record access through Telemetry.

GetRawData Parameters

Name	Description
titleId	The Asterian ID for your product. **Note:** A single platform adapter can support multiple products, this helps keep them distinct.
key	The unique key to identify the data to pull. Top-level collections will use the game account ID (<region><id>). Other collections will use a key defined by the parent item module.

Adding Collection Fields

After you create a Collection, you need to add fields that represent the columns to show in Atlas.

Note: Field Definitions for Items and Collections are the same.

Adding a Text Field

The most common field is the *Text* field. This field displays text-type data that is directly accessible to the platform adapter. Do not use this field if you need to convert the data coming from the Game Data API to text.

To add a Text field, add the following code at the end of the constructor.

```
FieldDefinitions.AddTextField("name")
                .WithCaption("Name")
                .WithFieldData(c => c.Name);


FieldDefinitions.AddTextField("level")
                .WithCaption("Level")
                .WithFieldData(c => c.Level.ToString());
```

The FieldDefinition property exposes a fluent interface for defining fields. The process starts by picking the method for the appropriate field type. The additional method *WithFieldData* is required; the other methods on the fluent interface are optional.

There are two important definition methods. Refer to our API documention for optional methods for each of the field types.

- *WithCaption*, applies a caption to the field. This caption can use special characters and spaces, and is distinct from the name, which can't. If there is no defined caption, the name will be used as the caption.
- *WithTheme*, applies a theme to the field. Themes produce style effects like color or italic text. The theme definition can be static, or it can be calculated from raw data. Building a custom theme requires additional work from our Product Design team.
  Note: The default available themes are still pending development, but will be made available when they are defined.

Adding a UI Link Field

The UI Link field is used to create a hyperlink to a different platform item. For example, switching from a character in a list to the details page for that character. When a GM clicks the link, the UI will transition to the new item page, and a bread crumb will display at the top of the window so the user can navigate back to the original page.

This field takes the text of the string plus a reference to load the new item screen. Replace **name** in the following code with the item link to load the data type.

```
FieldDefinitions.AddUiLinkField("name")
                .WithCaption("Name")
                .WithPathReference((titleId, character, key)=>
                    new PathReference(
                        "character",
                        $"{key};{character.ServerId}:{character.ServerId}",
                        titleId))
                .WithFieldData(c => c.Name);
```

Note: The format of the key for all data types other than *game* is determined by the platform adapter implementation. The only requirement is that the platform adapter can use this key to determine the unique data element.

Adding an Icon Field

The icon field, and its associated column, display a specific icon from a defined icon font. This field can use a custom font, but we use Angular Material fonts by default. If your product requires a custom font, it will require additional work from our Product Design team.

Icon fields look for custom fonts first, then the Atlas custom font, and finally the Angular Material font. If there is no icon, Atlas will display an unknown icon symbol.

Note: A custom font for Atlas is under development.

To add an Icon field add the following code to the end of the constructor.

```
FieldDefinitions.AddIconField("deleted")
                .WithCaption("")
                .WithFieldData(c => c.IsDeleted ? "trash" : "");
```

Note: An empty string represents *no icon* for if the icon should be conditional.

Additional Field Types

Our Support Platform API documentation contains additional field types that you can use in data type module factories.

Adding Commands

While Fields provide data to display in Atlas, Commands allow for manipulation of that data. Each module has the ability to store sets of commands to later expose in Atlas. These commands can apply to items and to rows in a collection. When used with a collection, commands can be single selections inline with row data, or multiple selections separate from rows.

Command Payloads

Every command requires a payload to provide its execution parameters. If they require the same parameters, multiple commands can access the same payload. Your client will send this payload, so it's important to keep two things in mind.

1. Even though Atlas is an internal-only tool, the payload should still only include data that's safe for GMs to view. Do not send private client data or, because the payload runs through Telemetry, personally identifiable information (PII) in the payload.
2. The payload should consist mostly of call parameters, not data to look up at execution.

The following example shows a character payload class.

```
public class CharacterPayload : CommandPayload
{
        public ulong ServerId { get; set; }
        public ulong CharacterId { get; set; }

        public override string ToString()
        {
            return $"Character {CharacterIdentifier}";
        }
}
```

Note: All payloads derive from the *CommandPayload* class. Multi-select commands use the *ToString* overload to help identify the row if the command isn't compatible with a particular multi-select.

Configuring the Command

After you create a payload, you can add the command to the collection module.

The following example shows a command that will add a 'delete character' command.

```
FieldDefinitions.AddCommand("DeleteCharacter", "Delete Character",
         c => new CharacterPayload {CharacterId = c.Data.CharacterId,
ServerId = c.Data.ServerId})
    .EnableMultiSelect()
    .ForTitleId(123)
    .Whenever(c => !c.IsDeleted)
    .WithConfirmationPrompt("Are you sure you want to delete the character
(s)?");
```

Commands always include a *commandID* that identifies the command for the platform, a client-side button prompt, and the payload definition. Each row in the data set will receive a command for both single- and multi-select commands.

The table to the right highlights some of the optional methods available in the fluent interface. The full list of commands is available in our API documentation.

Optional Methods

Name	Description
EnableMultiSelect	Used to determine if the command is single- or multi-select.
Whenever	This is a test method to determine if the command applies to a particular row.
WithConfirmationPrompt	If a prompt exists, the user needs to confrim the command after the prompt displays.
ForTitleId	Needed to lock a command to a particular title. If this entry is empty, the command responds to any title passed to the handler.

Handling the Command

After you configure the command, you need to write a handler to support the command's execution.

Continuing the example of a 'delete character' command, the following code executes that command and deletes the character.

```
public class DeleteCharacterCommandHandler :
CommandHandler<CharacterPayload>
    {
        private ICharacterService _characterService;

        public DeleteCharacterCommandHandler(ICharacterService
characterService)
            : base("DeleteCharacterAsync")
        {
            _characterService = characterService;
        }

        protected override async Task<CommandResponse> Handle(uint
atlasUserId, int? titleId, string? issueId, CharacterPayload payload)
        {
            var character = await _characterService.GetCharacterAsync
(payload.ServerId, payload.CharacterId);
            await _characterService.DeleteCharacterAsync(payload.ServerId,
payload.CharacterId);
            return new CommandResponse(true, $"Character {character.Name}
deleted.");
        }
    }
```

Note: Most optional parameters for the Handle method are for specific uses in Atlas. You can ignore most of these.

After a command runs successfully, it will generate a command response that contains information about the successful completion. Failed commands should lead to exceptions with an error message.

The command response parameters are shown in a table to the right

Command Response Parameters

Name	Description
requiresAudit	Indicates that Telemetry should record an audit for this command that contains the payload and the commandId. **Note:** We recommend leaving this set to true.
issueNote	If a note is left on the command, it will show in Atlas to indicate what operation was performed.
pathReferences	The same references used in the UI Link; these create links in Atlas GMs can click to see the recorded data. For example, if a character deletion is a 'soft delete', you can include a path reference to the deleted character in the Atlas issue.

Adding Additional Data Types

In previous examples we created a UI Link field for the character name.This link leads to a new data type—in this case, the character itself.

Creating a new data type is similar to the *game* data type. However, this data type would be specific to the platform adapter so there's no atlas.server level data type to store additional details.

This new data type uses many of the same field definitions as the others. In the following example, EnableMultiSelect is not included since it has no purpose on item-level commands. We've also modified the name to use a text field rather than a link.

Note: Item modules can contain UILinks if needed. For example, a link to a character's guild.

```
    public class CharacterItemModuleFactory : ItemModuleFactory<Character>
    {
        private readonly ICharacterService _characterService;

        public CharacterItemModuleFactory(ICharacterService characterService,
IEnumerable<ICollectionModuleFactory> collectionFactories)
            : base("character", "", true, collectionFactories)
        {
            _characterService = characterService;
            FieldDefinitions.AddTextField("name")
                .WithCaption("Name")
                .WithFieldData(c => c.Name);

            FieldDefinitions.AddTextField("level")
                .WithCaption("Level")
                .WithFieldData(c => c.Level.ToString());

            FieldDefinitions.AddIconField("deleted")
                .WithCaption("")
                .WithFieldData(c => c.IsDeleted ? "trash" : "");

            FieldDefinitions.AddCommand(
                    "DeleteCharacterAsync",
                    "Delete Character",
                    c =>
                        new CharacterPayload {CharacterId = c.Data.CharacterId, ServerId = c.Data.ServerId})
                .ForTitleId(123)
                .Whenever(c => !c.IsDeleted)
                .WithConfirmationPrompt("Are you sure you want to delete the character(s)?");

            AddTag("inventory", "widgets", "Inventory");
            AddTag("mail", "mail", "Mail");

            AddCollection("bags", "inventory");
            AddCollection("bank", "inventory");
            AddCollection("mail", "mail");
        }

        protected override Task<Character> GetRawDataAsync(int? titleId, string key)
        {
            var ids = key.Split(":");

            return _characterService.GetCharacterAsync(UInt64.Parse(ids[0]), UInt64.Parse(ids[1]));
        }
    }
```

Configuring Atlas

After creating the platform adapter, you need to deploy it through the Battle.Net API Gateway. Once deployed, you can register the platform adapter for its title through the Service Tech Platform Configuration.