A source-code repository as a starting point for API documentation

Good documentation emerges as a result of the tight collaboration of Subject Matter Experts (SMEs) and Technical Writers. Good API documentation, in particular, is an outcome of effective communication between the Development Team (Software Engineers, QAs, Architects) and Documentarians. One of the key aspects of effective communication is the ability of Technical Writers to ask specific practical questions during meetings. And to be able to prepare such questions, we need to get familiar with the API before communication starts - to dive into the context.

An excellent resource that can help you to obtain some preliminary idea about an API is the API's source-code repository. For documentarians without a technical background, it may sound quite unrealistic. However, in some cases, programming experience is not obligatory to read the code of an API.

One of such cases is the **Model-view-controller (MVC)** design pattern that is used to develop REST APIs with the **.Net platform**. In a nutshell, MVC implies processing some data (*models*) in specific methods (*controllers*) and presenting the results to the end-users (*views*). If you are new to the concept of MVC, this video gives an example that will help you to grasp the idea.

❗ Everything described in the article relates to .Net core 2.1+ versions.

In this article, we will see how a Technical Writer can recognize API components in particular code blocks. To do so, we will use a real GitLab repository of a **.Net service** that is accountable for managing Users' accounts – registration and authorization, editing contact details, etc. The service's REST API is implemented in **C#** using **the MVC** design pattern.

This article won't explain complicated technical aspects behind API development. After reading the article, you will (hopefully) be able to make HTTP requests to an MVC API without asking developers how to do it.

What exactly do we expect to find in a code repository?

First, the **basics**:

- the **number of endpoints**, their **types** and **addresses** (except for the hosting server's address),
- **request body parameters** and their **data types**,
- **response parameters**,
- possible **error messages**.

Also, you can try to figure out how one or another endpoint works (the **business logic** behind it). Although, in this case, you will most likely need some programming expertise.

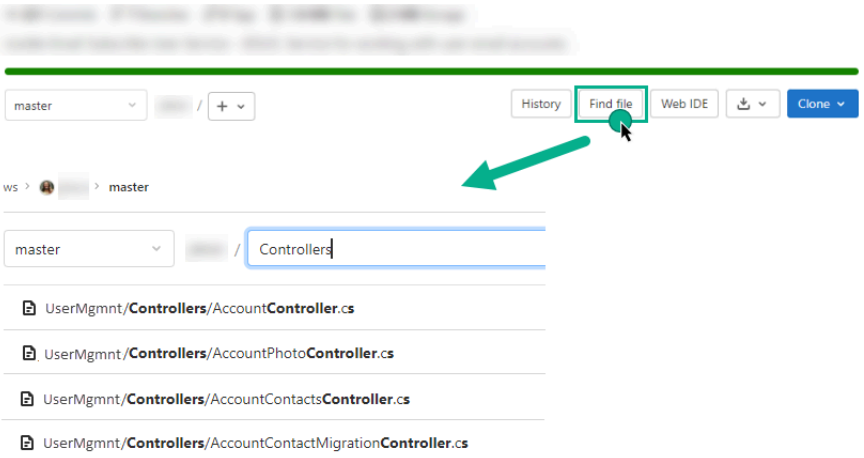
So, let's start our journey!

Where to search for?

In the MVC approach, APIs are implemented as a set of *controllers*. Each controller is a set of methods (endpoints) that implement a part of a service's business logic. The number of controllers depends on the service's complexity.

To learn something about API in a code repository, we should search for the files corresponding to controllers. Navigating through the complicated project structure that consists of dozens of folders and subfolders with non-obvious names does not seem like an easy task. But the good news is that developers tend to store the API code in a separate project folder named something like *API* or *Controllers,* while source code repositories allow searching by a file/folder name.

On the main page of our project's repository in GitLab, there is a button **Find file**. By clicking on it and entering a possible name of the folder with API controllers, we get the following list of files:



From the list, we can say that four controllers united in the self-titled folder constitute the service's API. From controllers' names, we can assume what their purpose is:

- *AccountController.cs* – processing Users' accounts (probably, registration and authorization),
- *AccountPhotoController.cs* – processing Users' photos (adding, removing, and editing accounts' photos),
- *AccountContactController.cs* – processing Users' contact details,
- *AccountContactMigrationController.cs* – presumably, a temporary service controller used once to migrate contact data from one storage to another.

When we open a controller's file and look at the names of its methods, we can prove or disprove the assumptions.

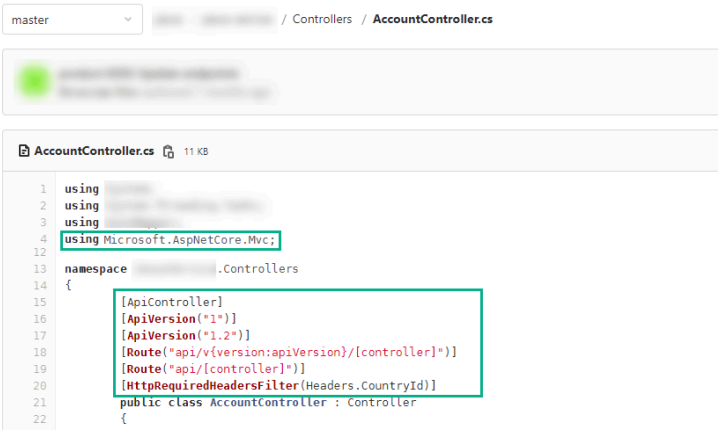
So, what we have **at this stage**:

- we can define how many logical parts the service's API has (by the number of controllers);
- we have the folder and the files necessary to discover API's endpoints.

Before getting to endpoints

In a specific controller's file, after the **namespace** statement, there may be a set of ***attributes*** – metadata identifiers of the controller enclosed in square brackets. These attributes provide information related to all the endpoints of the controller.

For example, the controller **AccountController** in our project has the following attributes:



- **[ApiController]** – this one serves development purposes as it allows automatic model validation in endpoints and other tricks useful at the development stage;
- **[ApiVersion("1")]** & **[ApiVersion("1.2")]** – currently, there are two supported versions of the API (1 and 1.2);
- **[Route("")]** – this part of the route is common for all the endpoints of the controller;
- **[HttpRequiredHeadersFilter()]** – when sending requests to the endpoints of this controller, we have to specify **Headers** given in the parentheses (in our case, the required header is **CountryId**).

It is far from a complete list of possible controller's attributes. There are more of them, and to find out what they are specified for in a particular controller, you need to google them or ask developers.

The common part of all endpoints' routes is specified in parentheses in the attribute **Route**. Let's take a look at the first Route attribute:

api/v{version:apiVersion}/[controller]

where
- instead of *{version:apiVersion}*, we put the needed API version – 1 or 1.2
- instead of *[controller]*, we put the controller's name, excluding the word Controller.

A few words about versions. In large projects, where several dozen end users leverage API, multiple API versions must be supported to enable a smooth transition from older versions to newer ones. In such projects, different API versions are represented in separate folders in the code repository. As Technical Writers, we have to pay attention and support documentation for all active v-s.

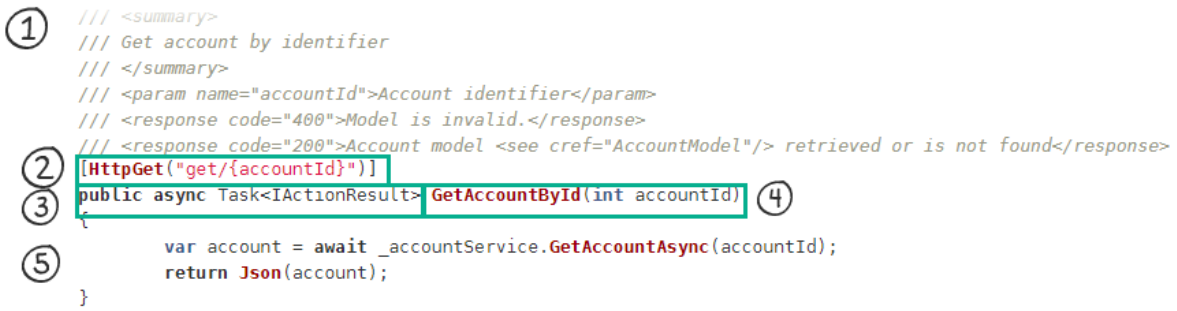
So, the common route part, according to the first Route attribute, may be **api/v1/Account** or **api/v1.2./Account**. According to the second Route attribute, it is **api/Account**. All the three work fine for HTTP requests.

Endpoints

As we already said, each controller is a set of endpoints that can be called by other software to get particular resources or perform specific actions.

A code block corresponding to an endpoint usually consists of the following components:

1. developers' comments (optionally)
2. an attribute indicating the endpoint's type and route
3. type of the endpoint's method
4. name of the endpoint's method with input parameters (request body parameters)
5. endpoint's business logic confined in curly braces




```
/// <summary>
/// Get account by identifier
/// </summary>
/// <param name="accountId">Account identifier</param>
/// <response code="400">Model is invalid.</response>
/// <response code="200">Account model <see cref="AccountModel"/> retrieved or is not found</response>
[HttpGet("get/{accountId}")]
public async Task<IActionResult> GetAccountById(int accountId)
{
        var account = await _accountService.GetAccountAsync(accountId);
        return Json(account);
}
```

Comments

Comments start with /// and provide an overview of an endpoint – its purpose, input parameters, response parameters, possible error messages.

In our example, the comments tell us the following information:

```
/// <summary>
/// Get account by identifier
/// </summary>
/// <param name="accountId">Account identifier</param>
/// <response code="400">Model is invalid.</response>
/// <response code="200">Account model <see cref="AccountModel"/> retrieved or is not found</response>
```

The endpoint gets information about a particular account by its identifier.

If an account with the specified id is found, the endpoint returns status 200 OK and the account's data in JSON format.

If an account with the specified id doesn't exist, the endpoint returns status 200 OK with the message "The model is not found".

The error code 400 is returned if the input model has a wrong format (for example, instead of a numeric value, an alphabetic one is specified).

Developers leave comments to automatically generate documentation for endpoints in Swagger. They don't do so all the time, but if they do, you don't need to read the article further. In fact, all the necessary information may be provided in the comments. So, always advocate writing comments!

Type and route

Type and route of an endpoint are specified as an attribute in square braces. The **type** (GET/POST/PUT/DELETE) is **always specified** in the method's attribute, while the **route may be omitted**.

If a controller has only one endpoint of the GET type, it's not necessary to design a separate route for it. In this case, the **full route** of the endpoint can be depicted as follows:

{RequestMethod} {serviceHostingAddress}/{commonRoute}

If a route is specified for an endpoint, it is important to remember that the route indicated for the endpoint is only **a part of the full route**. In this case, the **full route** can be depicted as follows:

{RequestMethod} {serviceHostingAddress}/{commonRoute}/{endpointRoute}

where

- *RequestMethod* - GET/POST/PUT/DELETE;
- *serviceHostingAddress* – address of the server where the service is hosted;
- *commonRoute* – the route specified in the attribute **[Route]** of the Controller (see the section Before getting to endpoints)
- *endpointRoute* – the unique part of the endpoint's root

The full route of an endpoint can be seen as a full address or an apartment: *serviceHostingAddress* is a city with many houses (endpoints), *commonRoute* is the number of the house where the apartment (endpoint) is, and *endpointRoute* is the apartment's unique number.

Let's assume that the Users' management service we consider in the article is hosted on a server with IP address 127.55.55.55, port 8000. So, in our case, *serviceHostingAddress* is **127.55.55.55:8000**.

From the section Before getting to endpoints, we remember that the common route may be **api/v1/Account**, **api/v1.2./Account**, or **api/Account**.

The attribute with the type and route of the endpoint tells us the unique part of the endpoint's route:

```
[HttpGet("get/{accountId}")]
```

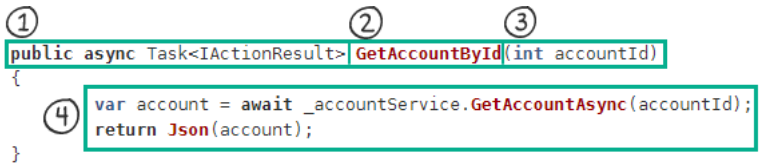
So, a request to call the endpoint looks like **GET *https:// 127.55.55.55:8000/api/Account/get/100,*** where 100 is the ID of the account that we want to get information for.

Endpoint's method. Type, input parameters, and business logic

What happens when an endpoint is called is defined in the corresponding method that implements the endpoint's business logic. An endpoint may be a specific digital location where particular resources may be found. Alternatively, it can perform some manipulations with data sent in the request body.

In our example, the endpoint returns data about a User. So, there is some code requesting the data from other services or directly from the database.

Using this example, we will take a look at the endpoint's method's structure:



An endpoint's method consists of four parts:

1 – method's type;
2 – method's name;
3 – input parameters;
4 – method's body (lines of code to execute to get the endpoint's ultimate result).

Method's type defines what the method returns, and we will touch upon it in the section Response parameters and Error messages.

Method's name is a paraphrased endpoint's route and says what the method does.

Input parameters specified in parentheses after the method's name can be of different types - those taken from the method's route, request body, request URL, request headers, etc. We will take a closer look at the first two of them:

*type 1 - taken from the **method's route***

- parameters are specified in curly braces in the method's route
- in parentheses after the method's name, the parameters are specified without any attributes

```
[HttpGet("get/{accountId}")]
public async Task<IActionResult> GetAccountById(int accountId)
```

- such parameters are usually of standard data types (integer, string, char, etc.)

*type 2 - taken from the **request body***

- parameters are not specified in curly braces in the method's route
- in parentheses after the method's name, the parameters are specified with the **[FromBody]** attribute

```
[HttpPost("auth")]
public async Task<IActionResult> Authenticate([FromBody] AuthenticationDto authenticationModel)
```

- such parameters are usually of custom data types and represented as *Data Transfer Objects*

While the first type is quite intuitive, the second one may be tricky. We will discuss it in detail later in the article after speaking about Data types and DTOs in the next section.

Data types and DTOs

DTOs

APIs allow independent services to communicate with each other by calling corresponding endpoints. In some cases, one piece of software requests data from another one, and no data processing is involved in the communication. For example, an endpoint that returns a list of all active Users in the system. For such endpoints, methods that implement their business logic do not take any input parameters, and there is no request body.

In other cases, one service may want the other to perform certain operations with the initial set of parameters and return the modified set of parameters. For example, an endpoint that takes the registration data of a User, creates a new account for the User, sets the active status to the account, and returns the account's id and status. Methods for these endpoints take input parameters sent in the request body.

When there are one-two input parameters, they can be specified in braces after a method's name separated by a comma:

```
[HttpGet("authenticate")]
public async Task<IActionResult> Authenticate(int accountId, string password)
```

However, when we need to send three or more parameters to an endpoint, listing them with commas doesn't look like a convenient way to write code, let alone maintain it in future. As a solution, developers unite all the parameters that an endpoint takes into a single structure – a data transfer object. Compare the two options:

without DTO

```
[HttpGet("register")]
public async Task<IActionResult> Register([FromBody] string userName,
                                          [FromBody] string userEmail,
                                          [FromBody] string userPassword,
                                          [FromBody] string confirmedPassword,
                                          [FromBody] byte RegistrationSource)
```

with DTO

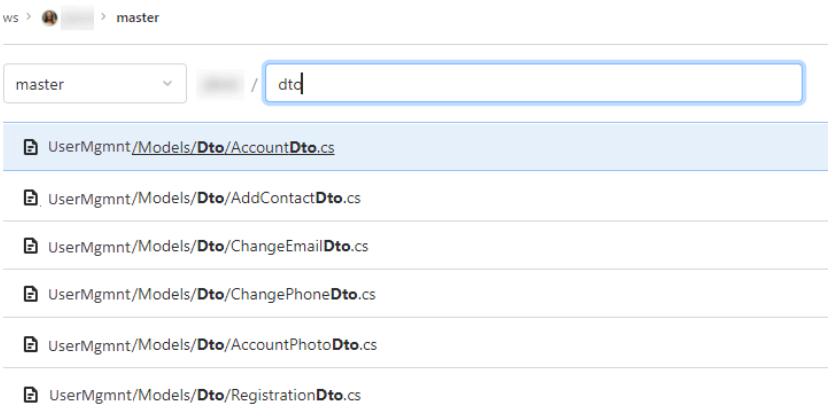
```
[HttpGet("register")]
public async Task<IActionResult> Register([FromBody] RegistrationDto registrationModel)
```

```
namespace Models.Dto
{
    public class RegistrationDto
    {
        public string userName { get; set; }

        public string userEmail { get; set; }

        public string userPassword { get; set; }

        public string confirmedPassword { get; set; }

        public byte RegistrationSource { get; set; }
    }
}
```

A **Data Transfer Object (DTO)** is an object that stores all necessary parameters to send from one application to another. In the MVC design pattern, a DTO is a **model**. A DTO is a contract that defines which parameters an endpoint expects in the request body.

A DTO is a separate C# class (a .cs file), and all DTOs (all the .cs files) of a service's API are usually stored in a separate self-titled folder (the folder may be called *Models* or *DataTypes*).

In the project that we are considering, developers agreed to store DTOs in the **Dto** folder and add the prefix **-Dto** to the file names. Knowing the agreement for dealing with models in the project, we can find them in the project's repository by clicking on the button **Find file** and entering the name of the folder with DTOs:



When opening a DTO file, we can see a set of parameters. When an endpoint takes a particular DTO as an input parameter, these parameters should be specified in the request body. For example, the registration endpoint from the example above takes **RegistrationDto**:

DTO:

```
namespace Models.Dto
{
    public class RegistrationDto
    {
        public string userName { get; set; }

        public string userEmail { get; set; }

        public string userPassword { get; set; }

        public string confirmedPassword { get; set; }

        public byte RegistrationSource { get; set; }
    }
}
```

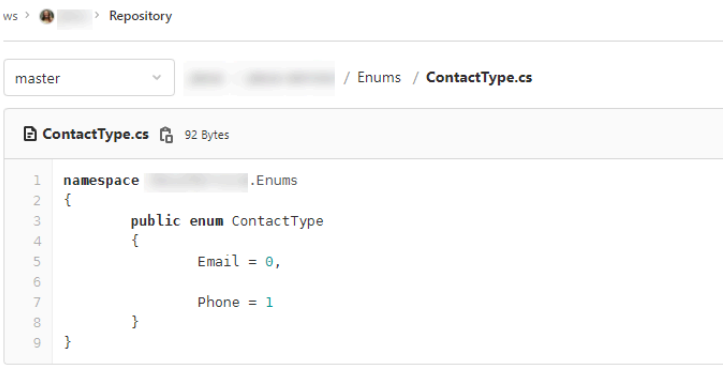
Corresponding request body:

```
{
    "userName": "John Smith",
    "userEmail": "smithj@test.com",
    "userPassword": "12345",
    "confirmedPassword": "12345",
    "RegistrationSource": "1"
}
```

Data types

Input parameters in endpoints' methods can be of a **built-in data type** (string, boolean, integer, etc.) or a **custom data type**. Developers "invent" custom data types to define logical groups of business terms.

For example, the service for authorization and registration operates with contact details of two types – emails and phone numbers. In future, other contact types may be added to the business rules. To simplify working with this classification, a new custom data type **ContactType** may be created:



Often, such custom data types are **enums** – sets of possible values of the data type with integer identifiers assigned to each value. They are separate **.cs** files (with the word *Type* or *Flags* in their names), united in a separate folder (*Enums*, *CustomDataTypes*, etc.).

A DTO is also a custom data type. However, it is not a set of possible values of a particular business term. It is a set of related parameters organized in a convenient logical structure.

Request body parameters

Input parameters specified in an endpoint's method with the **[FromBody]** attribute before their data type are sent in the request body. Several parameters may be listed with a comma or just one parameter representing a particular custom data transfer object.

In the first case, we see the list of parameters and form a JSON structure from it.
For example:

```
1 [HttpPost("simpleAuth")]
2 public async Task<IActionResult> SimpleAuthenticate([FromBody] string email, [FromBody] string password)
3 {
4     // authentication procedure
5     return Ok();
6 }
```

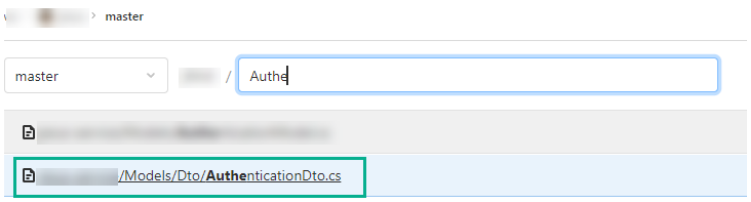
Here, we have two input parameters coming from the request body - string email and string password. Transforming them into JSON format, we get the following request body for the endpoint:

```
1 {
2     "email": "smithj@test.com",
3     "password": "12345"
4 }
```

When an input parameter is an object of a custom data type, we first need to find the definition of the custom data type among DTOs of the service's API. We search for a separate .cs file (class) named like the data type.
For example:

```
1 [HttpPost("auth")]
2 public async Task<IActionResult> Authenticate([FromBody] AuthenticationDto authenticationModel)
3 {
4     // authentication procedure
5     return Ok();
6 }
```

Here, we have an input parameter of the AuthenticationDto type, so we search for a file named AuthenticationDto.cs where this data type is defined:



When we open the file, we can see what parameters define the data type:

```
using UserMgmnt.Enums;

namespace UserMgmnt.Models.Dto
{
    public class AuthenticationDto
    {
        /// <summary>
        /// Gets or sets User's email
        /// </summary>
        public string Email { get; set; }

        /// <summary>
        /// Gets or sets User's password
        /// </summary>
        [MinLength(5)]
        public string Password { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether authentication was from mobile (true) or desktop (false)
        /// </summary>
        [Required]
        public bool? IsMobile { get; set; }

        /// <summary>
        /// Gets or sets the login type
        /// </summary>
        [Required]
        public LoginType? LoginType { get; set; }
    }
}
```

AuthenticationDto is a set of four parameters:

1. string *email* - User's email address;
2. string *password* - User's password;
3. bool *IsMobile* - whether the User authenticates from the Mobile interface (true) or not (false);
4. LoginType *LoginType* - a custom type that defines how exactly the User is authenticated.

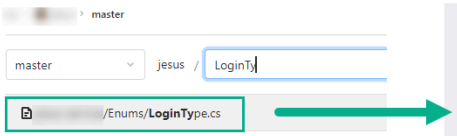
Attributes specified before some parameters in the DTO serve for validation:

- **[Required]** means that if the parameter is omitted, the endpoint returns a validation error;
- **[MinLength(n)]** specifies the minimum acceptable length for the parameter.
  Otherwise, a validation error occurs.

These are just two examples, but many more attributes can be specified before for parameters in DTOs.

The question mark **(?)** after data type indicates that the parameter can be nullable or omitted.

LoginType is another custom data type, so we need to search for its definition as well:




```cs
namespace UserMgmnt.Enums
{
   public enum LoginType
   {
      /// <summary>
      /// default registration/authentication
      /// </summary>
      Default = 0,

      /// <summary>
      /// via Facebook account
      /// </summary>
      Facebook = 1,

      /// <summary>
      /// via LinkedIn account
      /// </summary>
      LinkedIn = 2,

      /// <summary>
      /// vis Google account
      /// </summary>
      Google = 5
   }
}
```

LoginType is an enum with possible values 0, 1, 2, and 5.

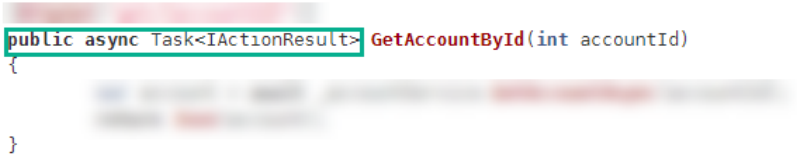
Bringing the above together, we form the following JSON request body for the endpoint:

```json
{
    "Email": "smithj@test.com",
    "Password": "12345",
    "IsMobile": true,
    "LoginType": "0"
}
```

Response parameters and Error messages

We reached the point when we can send requests to endpoints using API testing software like Postman. Although looking for response parameters and possible error messages in code may be excessive (we can figure them out from responses), it still can provide valuable and non-obvious details.

Whether an endpoint returns a parameter or not, the data type of the returned parameter, and error messages you may get - all of these depend on the endpoint's method's type. Let's consider this term in detail on the example of our endpoint for getting information about a User:



The first thing we see is that the method is **public.** It is an **access modifier** indicating that the endpoint is accessible from any external services, not just within the controller where it is defined.

The next thing we discover is that the method is an asynchronous **task** (async Task) which means that when multiple HTTP requests are made, they will be processed asynchronously. For now, suffice it to say that in MVC, it is a common practice to implement endpoints as asynchronous tasks.

The method's type (**<IActionResult>**) tells us what type of objects the method returns. In ASP.NET Core MVC pattern, there are three possible return types - **Specific type, IActionResult, and ActionResult<T>**. I won't try to explain the difference between them (it has to do with technical implementation), but even without understanding technical specifics, we still can take a look at possible returned results.

In **success cases**, when the input request is processed by the endpoint, the returning result may be:

- any *2XX status code* with or without a set of parameters implied by the returning model:

```
[HttpGet]
public List<User> Get() =>
    _repository.GetUsers(); // will return 200 OK and a list of Users in JSON format
```

- *return OK()* - status code *200 OK* with no parameters;
- *return OK(result)* - status code *200 OK* with the value of the *result* variable;
- *return Content("Registration is successful")* - status code *200 OK* with the corresponding explanatory message;
- *return Ok(_users.List())* - status code *200 OK* with the list of Users in JSON format

```
[HttpGet]
public IActionResult GetUsers()
{
    var users = _repository.GetUsers();
    return OK(_users.List());
}
```

- *file, content* (plain text), or *redirection to another action*:

In **error cases**, when the endpoint cannot process the input request for some reason, the returning result may be:

- *return BadRequest()* - status code *400 Bad Request* with no explanatory message;
- *return BadRequest("The input parameter has a wrong format")* - status code *400 Bad Request* with the corresponding explanatory message;

```
[HttpPost]
public IActionResult RegisterUser([FromBody] UserRegisterModel userRegModel)
{
    if (ModelState.IsValid)
    {
        // .. perform registration
        return Ok(result);
    }
    return BadRequest(ModelState);
}
```

- *return NotFound("User not found")* - status code *404 Not found* with the corresponding explanatory message;
- *4XX status codes* are returned if there are issues with input data (on the sender's side);
- *5XX status codes* are returned if the receiving server cannot process requests

Sometimes, exceptions are handled using *try-catch statements* or other error handling strategies. In such cases, better ask developers to explain to you the mechanism so that you will be able to reproduce error cases.

In most cases, look for the keyword ***return*** to figure out what exactly you should expect to get from the endpoint.

Business logic

Business logic implemented in an endpoint's method can be tricky enough even for developers. There may be a lot of external services involved in data processing, complicated conditional branches, and numerous loops.

Explaining how to navigate through code lines trying to figure out how things work would be out of the scope of the article. At the starting point of getting familiar with an API, diving deep into business logic is inefficient, though it may be pretty gripping.

A more rational option is to ask developers to show you the code while answering your questions about the API. You would be able to get back to particular code blocks when working on the API documentation.

More examples

Taking into account what we discussed earlier, let's take a look at another controller of the service - **AccountContactsController** - and try to elicit some information about its two endpoints:

```
1  using System.Threading.Tasks;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace [redacted].Controllers
5  {
6      [ApiController]
7      [ApiVersion("1.2")]
8      [Route("api/v{version:apiVersion}/[controller]")]
9      [HttpRequiredHeadersFilter(Headers.CountryId)]
10     public class AccountContactsController : Controller
11     {
12         /// class constructors
13
14         /// <summary>
15         /// Adds a new contact for the specified user
16         /// </summary>
17         /// <param name="model">A model representing new contact</param>
18         /// <response code="400">Input AddContact model is invalid</response>
19         /// <response code="200">Adding a new contact for the account is completed</response>
20         [HttpPost]
21         public async Task<IActionResult> AddContact([FromBody] AddContactDto model)
22         {
23             // add the contact to the User's account
24
25             return Ok(result);
26         }
27
28         /// <summary>
29         /// Deletes a contact for the specified user
30         /// </summary>
31         /// <param name="model">A model representing a contact to be deleted</param>
32         /// <response code="400">Input DeleteContact model is invalid</response>
33         /// <response code="200">Deletion of contact for the account is completed</response>
34         [HttpDelete]
35         public async Task<IActionResult> DeleteContact([FromBody] DeleteContactDto model)
36         {
37             // delete the contact from the User's account
38
39             return Ok(result);
40         }
41     }
42 }
```

We will mark the code sections with numbers to refer to them later:

- **1** - controller's attributes;
- **2** - the first endpoint;
- **3** - the second endpoint.

POST endpoint

There is only one POST endpoint of the controller, and it does not have a unique route. Its route may be derived from the controller's attributes (1):

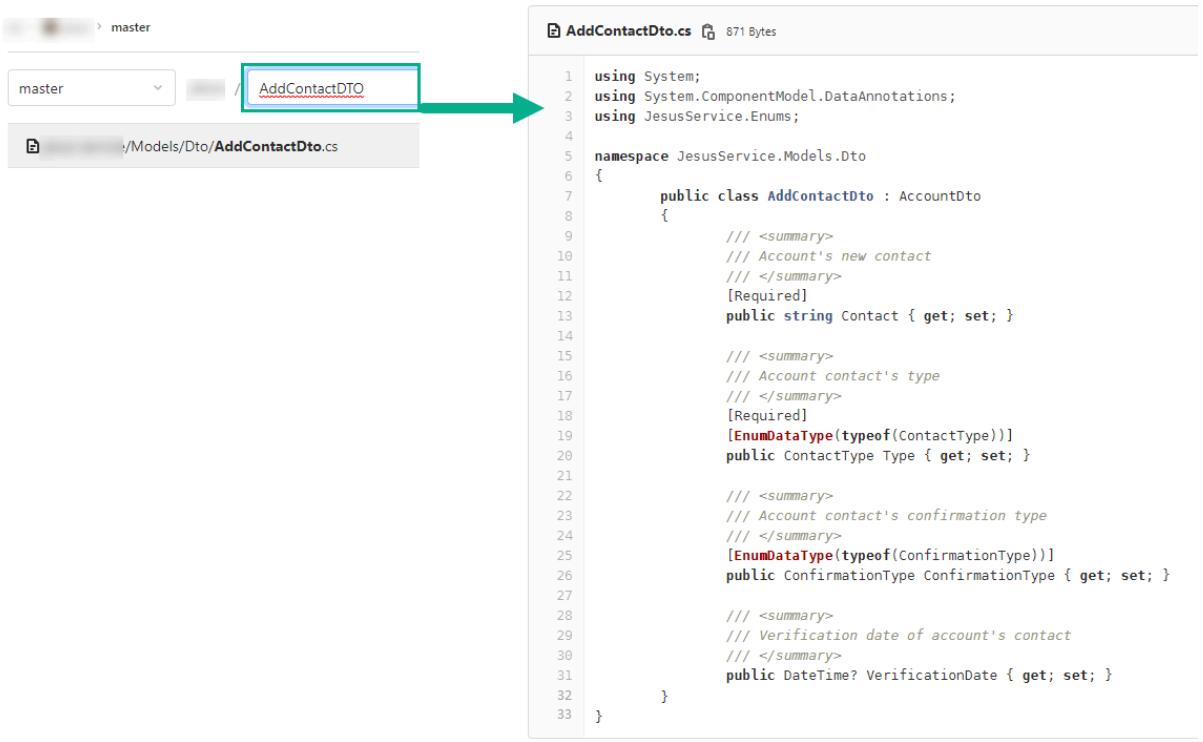
POST {*serviceHostingAddress*}/api/v1.2/AccountContacts

where *serviceHostingAddress* should be clarified with developers.

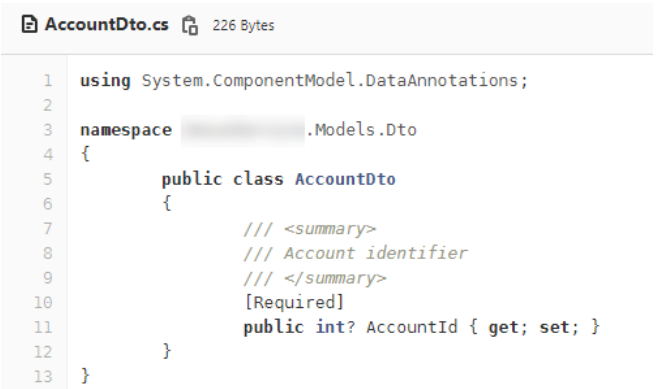
Also, from (1), we see that the endpoint requires specifying **CountryId in the request headers**. We will keep it in mind.

From the comments in (2), we know that the endpoint adds a contact to the specified User's account. It returns status code **200 OK** if the contact is added successfully and **400 BadRequest** if the input model has a wrong format.

As an **input parameter**, the endpoint takes an **AddContactDTO**, the definition of which we can find in the repository:



The first thing we notice is that the DTO inherits from another DTO **AccountDTO**, the definition of which looks like:

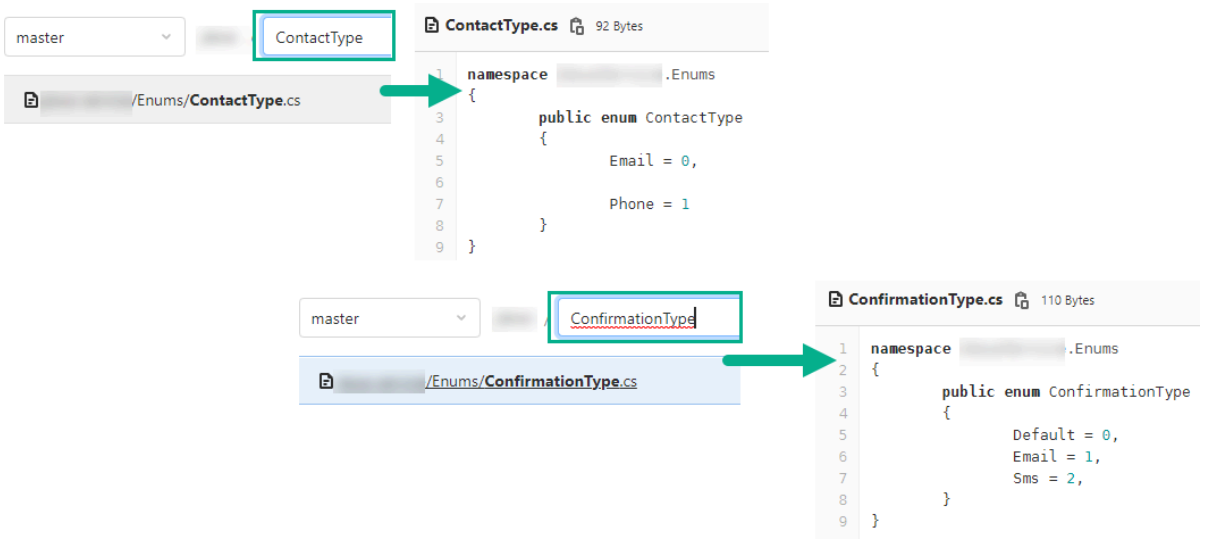


Here, **AccountId** is a required integer parameter that indicates the ID of the account to which the new contact will be added.

The **AddContactDTO** consists of four parameters:

- string **Contact** - User's contact itself (value of the phone number/email address/etc.), a required text parameter with no restrictions on length;
- ContactType **Type** - a type of the User's contact, a required parameter that can take values of the custom enum data type **ContactType;**
- ConfirmationType **ConfirmationType** - what type of confirmation the User used to add the contact to the account, a non-required parameter that can take values of the custom enum data type **ConfirmationType;**
- DateTime **VerificationDate** - date and time when the contact was verified; non-required parameter which takes null if omitted (the question mark after DateTime points it out).

Let's find definitions for **ContactType** and **ConfirmationType** enums:



Bringing together the parameters from the two DTOs, we can now compose a request body for the request in the JSON format:

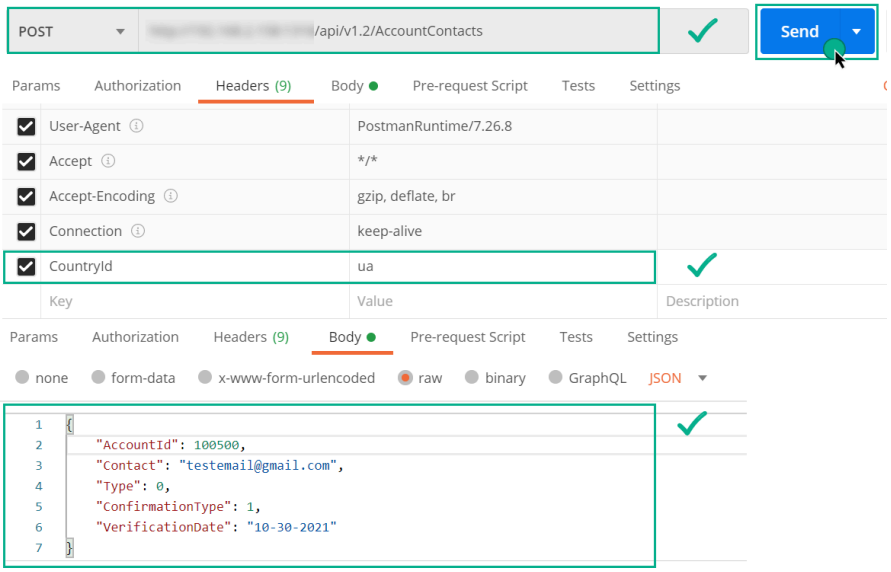
```
{
    "AccountId": 100500,
    "Contact": "testemail@gmail.com",
    "Type": 0,
    "ConfirmationType": 1,
    "VerificationDate": 10-30-2021
}
```

Speaking about returned parameter, the endpoint returns the status code ***200 OK*** and the value of the **result** variable (2):

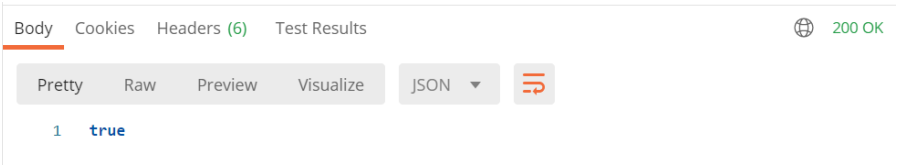
```
return Ok(result);
```

The type of the variable **result** may be derived from the business logic of the method (which we commented on as excessive details in this article). Another option is to test the request and get the return parameter experimentally.

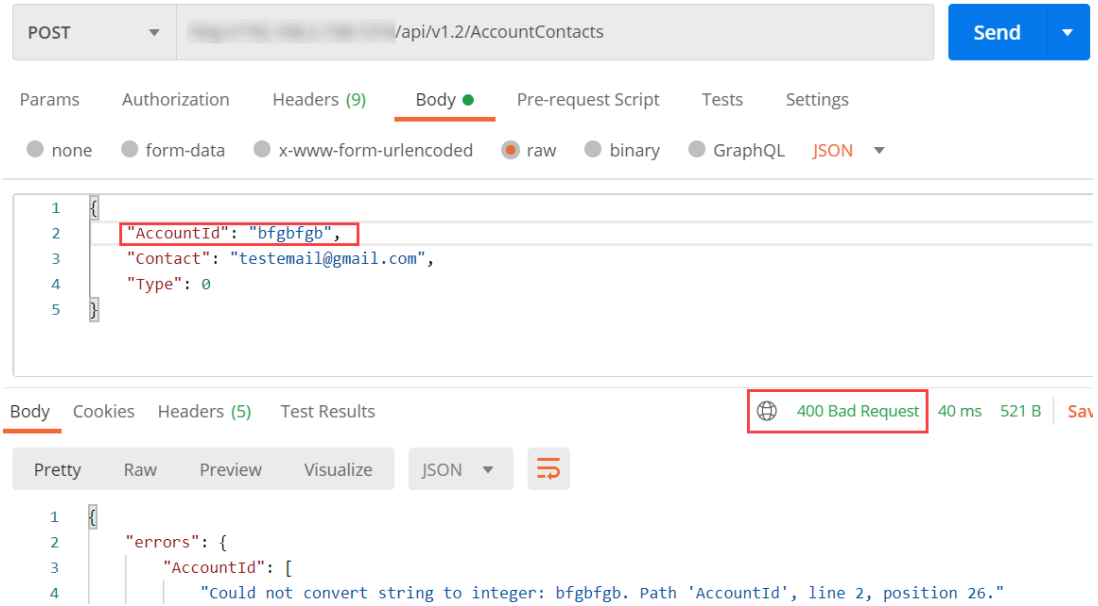
So, we now can create an HTTP request to test the endpoint using, for example, Postman:



After sending the request, we get in return:



To get error messages, we can modify the input model so that some of its parameters are missed or have a wrong format:



DELETE endpoint

Only one DELETE endpoint exists in the controller, so we derive its route from the controller's attributes (1):

DELETE {*serviceHostingAddress*}/api/v1.2/AccountContacts

From (1), we remember about **CountryId** required **in the request headers**.

From the comments in (3), we know that the endpoint deletes a contact from the specified User's account. It returns status code **200 OK** if the contact is deleted successfully and **400 BadRequest** if the input model has a wrong format.

As an **input parameter**, the endpoint takes a **DeleteContactDTO**, the definition of which we can find in the repository:

This DTO also inherits from the **AccountDto** and has two own parameters:

- string **Contact** - User's contact itself (value of the phone number/email address/etc.), a required text parameter with no restrictions on length;
- ContactType **Type** - a type of the User's contact, a required parameter that can take values of the custom enum data type **ContactType**.

Bringing together the parameters from the two DTOs, we can now compose a request body for the request in the JSON format:

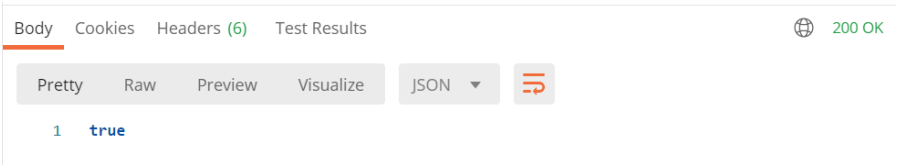
```
{
    "AccountId": 100500,
    "Contact": "testemail@gmail.com",
    "Type": 0
}
```

The endpoint returns status code ***200 OK*** and the value of the **result** variable (3):

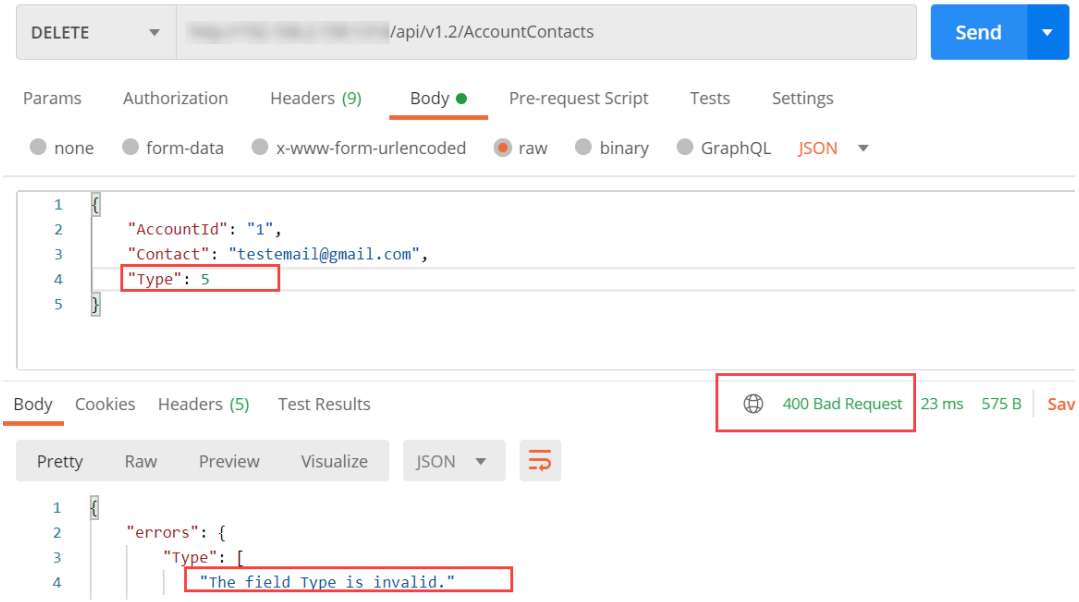
```
return Ok(result);
```

So, we now can create an HTTP request to test the endpoint in Postman:

After sending the request, we get in return:



Let's try to get an error:



To sum up

This article aims to demonstrate that by looking at a code repository of a REST API, it's possible to derive information needed to test its endpoints without involving developers at the preliminary stage. We deliberately missed deep technical details about the controller's attributes, response parameters, error messages and other API development aspects not to avert Documentarians with no or little development experience from navigating in the code.

Hopefully, after reading the article, when documenting an API, you will have enough confidence to access the code repository and discover the API's endpoints yourself.

P.S. Keeping you up to date with the API updates

If you don't want to rely on developers to update you about changes to the API, subscribing to the API's RSS feed will keep you in the loop.

For example, I use RSSOwl to receive notifications about recent updates in the codebase.

I will show you how to subscribe to an RSS feed of a GitLab repository in RSSOwl.

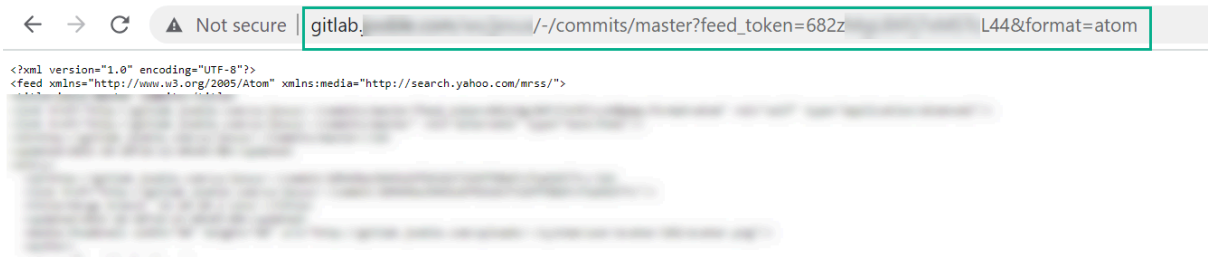
Prerequisites

- RSSOwl installed
- Access to the GitLab repository

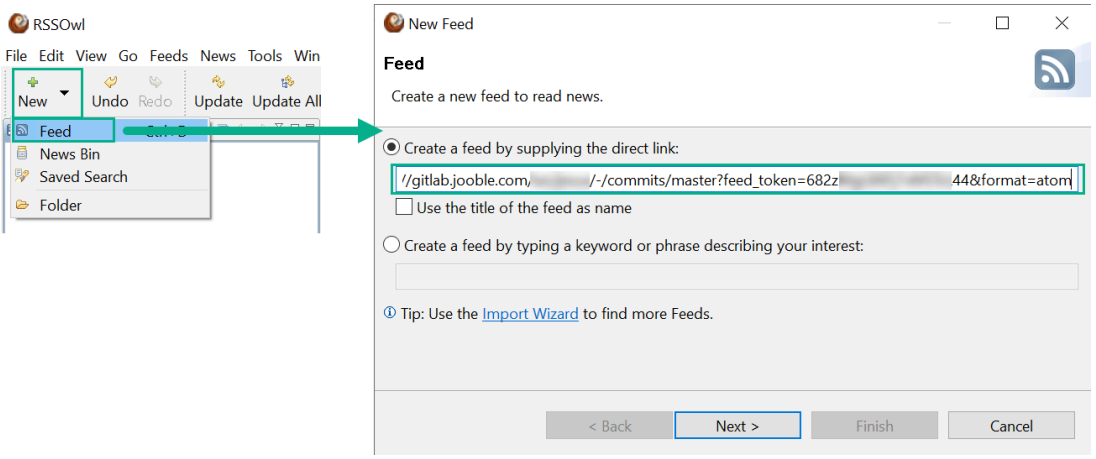
Subscribing

1. Go to the GitLab repository, **Repository -> Commits** tab and click on the RSS button in the top right corner of the page:

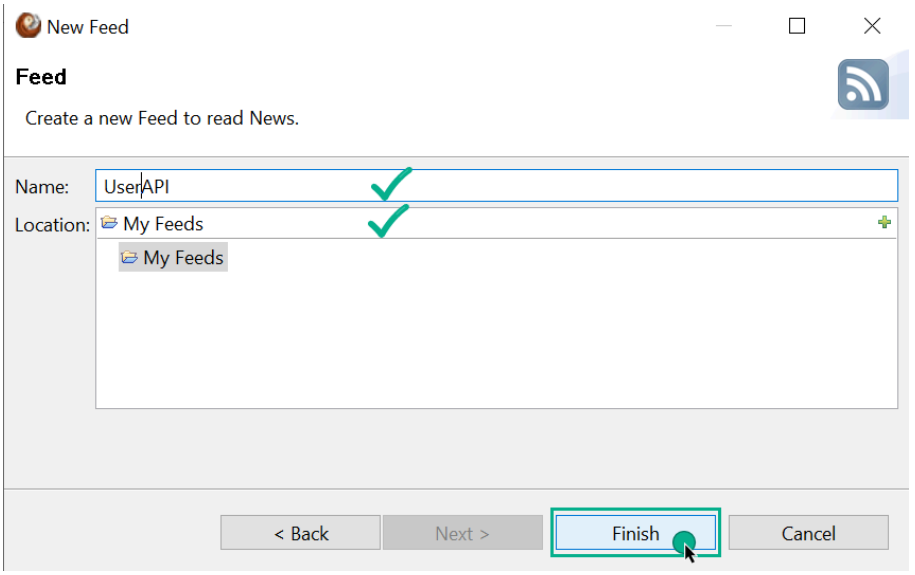
2. Copy the link of the opened web page:



3. Open RSSOwl, click on the button **New** and select the **Feed** option. In the popup window, specify the link you copied in the previous step in the input field after **Create a feed by supplying the direct link**:



4. In the next popup window, specify the **Name** and **Location** of the feed and click on the button **Finish**:



Now, you will receive notifications about new merge requests to the master branch of the API.