Python Web Channel

Release 1.9

Cihan Uyanik

Jan 02, 2024

CONTENTS:

`pywebchannel` is a tool that automatically generates TypeScript files for QWebChannel Python local backend. It allows you to create a stunning UI for your Python project using web technologies such as HTML, CSS, and JavaScript.

With `pywebchannel`, you can:

- Write your backend logic in Python and use Qt (PySide6) for communication.
- Use QWebChannel to communicate with the web frontend and expose your Python objects and methods.
- Write your frontend UI in any web framework of your choice, such as vanilla JS, React, Solid, Vue, etc.
- Enjoy the benefits of TypeScript, such as type safety, code completion, and error detection.
- Save time and effort by automatically generating TypeScript interfaces from your Python code.

Thank you for your interest in `pywebchannel`. I hope you enjoy using it as much as I enjoyed creating it.

CHAPTER

ONE

TYPE-SCRIPT GENERATOR

The TypeScript Generator part of this library has a file watcher that translates Python code to TypeScript interfaces. This enables safe and easy communication between your Python backend and your desired frontend (vanilla JS, React, Solid, Vue, etc.). To use the TypeScript Generator, run the `ts_generator.py` script and specify the folders that contain the Python files you would like watch for auto-instant conversion.

CHAPTER

TWO

CONTROLLER UTILITIES

pywebchannel provides helpful classes, functions and decorators to generate proper controller classes which can be exposed to a UI written by using web technologies. All given types are self documented and easy to follow.

2.1 What is Python Web Channel

2.1.1 Python Web Channel

pywebchannel is a tool that automatically generates TypeScript files for QWebChannel Python local backend. It allows you to create a stunning UI for your Python project using web technologies such as HTML, CSS, and JavaScript.

With pywebchannel, you can:

- Write your backend logic in Python and use Qt (PySide6) for communication.
- Use QWebChannel to communicate with the web frontend and expose your Python objects and methods.
- Write your frontend UI in any web framework of your choice, such as vanilla JS, React, Solid, Vue, etc.
- Enjoy the benefits of TypeScript, such as type safety, code completion, and error detection.
- Save time and effort by automatically generating TypeScript interfaces from your Python code.

2.1.2 Type-Script Generator

The TypeScript Generator part of this library has a file watcher that translates Python code to TypeScript interfaces. This enables safe and easy communication between your Python backend and your desired frontend (vanilla JS, React, Solid, Vue, etc.). To use the TypeScript Generator, run the ts_generator.py script and specify the folders that contain the Python files you would like watch for auto-instant conversion.

2.1.3 Controller Utilities

pywebchannel provides helpful classes, functions and decorators to generate proper controller classes which can be exposed to a UI written by using web technologies. All given types are self documented and easy to follow.

Documentation & API

2.2 Why do you need this?

You want to create a professional UI with modern web technologies and python. But you face many challenges with the existing libraries. Some of them rely on `window` manipulation, which is not compatible with most web frameworks. Some of them use RestAPI libraries, which add a lot of overhead, complexity and state management issues. Some of them use `WebSockets` only for function calls, without any real-time synchronization features.

Among these, `QWebChannel` seems to be the best option, with a lightweight `WebSocket` protocol and features like synchronization, function calls and property access. However, it also has its own limitations, mainly related to the Qt type system. You cannot use it with non-supported Qt types, without doing complex conversions, manual type adjustments and boiler-plate code, just to satisfy Qt. This makes development difficult and error-prone. And even if you manage to do that on the python side, you still have to deal with a frontend development cycle, with no type-hinting, no auto-completion, no compile time validation etc., which is a nightmare in javascript environment.

But don't worry, `pywebchannel` library is here to solve your problem .

2.2.1 Let's investigate the problems together

Suppose you want to build a `todo application` with python and web technologies. You will require some functionality to store the data (list of todos), modify it (add, remove, update), and inform the frontend about the changes.

2.2.2 Signals

You want to create a notification mechanism to the frontend, when you add a new todo item. You can use `QtCore.Signals` for that.

```
# Inside your controller class
new_todo_added = QtCore.Signal()
```

This signal can be emitted in your python code, after adding the item to your list. And your frontend will receive it, if it is connected to `new_todo_added` signal. But this signal does not carry any information about the new item. How can you send some data with it?

```
# Inside your controller class
new_todo_added = QtCore.Signal(str)
```

This signal can be emitted with a string parameter, and your frontend will receive it. For instance, if your todos have an `id` field of type `str`, you can emit it. `str` is a supported type in Qt, so it works. But what if you want to send a `Todo` object, which is a custom object of yours that inherits from `pydantic.BaseModel`?

```
# Inside your controller class
new_todo_added = QtCore.Signal(Todo)
```

This will cause an exception like this:

```
TypeError: Signal must be bound to a QObject, not 'Todo'
```

This is because, `Todo` is not a supported type in Qt. You can use `QtCore.QObject` as a base class for your `Todo`, to avoid this error. But then, you will face another problem, which is not even caught by exception mechanism. You will get an `empty object` in your frontend, instead of a `Todo` object. This is because Qt does not know how to serialize your `Todo` object to a valid json object. The simplest way to make it work is to use `dict` instead of `Todo` object.

```
new_todo_added = QtCore.Signal(dict)
```

But then, you will lose all the type information, and need to do type conversions. I don't even need to mention about `lists`. You need to take care of all these details, and keep your frontend and backend in sync. This is too much hassle…

You can use pywebchannel library, and define your signal like this:

with a list of types:

```
from pywebchannel import Signal

# Inside your controller class
new_todo_added = Signal([Todo])
```

or even better, with argument dictionary in the form of `{arg1_name:  arg1_type, ...}`:

```
from pywebchannel import Signal

# Inside your controller class
new_todo_added = Signal({'new_todo': Todo})
```

This will ensure that Qt is happy, and your frontend and backend are in sync.

And this is just the tip of the iceberg .

2.2.3 Properties

A property is a way to access and modify an internal (usually private) variable, with a getter and setter, in your class. It is a common feature in object-oriented programming. The benefits of using properties in Qt or PySide are that, you can create a signal for a property, so that any listeners or connected objects will be updated when the property changes.

For example, you have a property that keeps track of the number of todos. I know it is silly, but it is just for illustration.

```
# Inside your controller class
todoCount = QtCore.Property(int)
```

This is how you want to write your code. And also, you want to have a signal, that is triggered when the value of todoCount changes, you can call that signal something like `todoCountChanged`.

But that is not possible. You have to define a getter and setter for your property, and also a signal for it.

```
# Inside your controller class

def __init__(self):
    # You need a back variable to hold the value of your property
    self._todoCount = 0


# You need a signal to notify
todoCountChanged = QtCore.Signal(int, arguments=['todoCount'])


# You need a getter
def get_todoCount(self) -> int:
```

(continues on next page)

(continued from previous page)

```
    return self._todoCount


# You need a setter
def set_todoCount(self, value: int):
    if self._todoCount != value:
        self._todoCount = value
        self.todoCountChanged.emit(value)


# And finally, you can define your property
todoCount = QtCore.Property(int, fget=get_todoCount, fset=set_todoCount,␣
↪notify=todoCountChanged)
```

What the f… is this?

I don't even want to talk about the type conversions mentioned in `Signals` section. You have to do all these things for Properties too.

Instead of this sh…, you can use `pywebchannel` library, and define your property like this:

```
from pywebchannel import Property

# Inside your controller class
todoCount = Property(int, init_val=0)
```

And that's it. This will:

- ensure that Qt is happy, and your frontend and backend are in sync.
- create a private variable called `_todoCount` to store the value of your property.
- create a getter and setter for you as exactly written above.
- create a signal called `todoCountChanged`

If you want to have a different implementation for your getter and setter, you can still define one or both of them, and pass it as an argument to `Property`.

2.2.4 Actions

Actions are functions that you can call from your frontend. You can create an action in PySide like this:

```
# Inside your controller class
@QtCore.Slot(str)
def sayHello(self, name: str):
    # Do something with todo
    pass
```

This works, and you can call this function from your frontend. But the type issues mentioned above are still there.

```
# Inside your controller class
@QtCore.Slot(Todo)
def addTodo(self, todo: Todo):
    # Do something with todo
    pass
```

This does not work, and you will not even get an exception about that. Your function will be called with an empty argument . Most likely your application will crash, and your frontend will not even know why.

- This is because of one of the input arguments. You have to consider all the input arguments, and make sure that your frontend and backend are in sync.
- Return values also have the same problem, 'type matching' and 'keeping' Qt and serialization happy.
- If you want to notify the frontend about the execution result, you have to create your own signal, and emit it.
- You also have to handle exceptions as well.

You will end up with a lot of boilerplate code, which is not even related to your business logic.

```python
# Inside your controller class

# Create a signal for notification
new_todo_added = QtCore.Signal(dict, arguments=['new_todo'])


# Create a slot for your action
@QtCore.Slot(dict, result=dict)
def addTodo(self, todo: dict):
    try:
        todoObj = Todo.parse_obj(todo)

        # Do something with todo

        self.new_todo_added.emit(todoObj.dict())

        return {'success': True,
                'error': None,
                'data': todoObj.dict()}
    except ParseError as e:
        return {'success': False,
                'error': f"Invalid todo object: {e}",
                'data': None}
    except Exception as e:
        return {'success': False,
                'error': f"Unknown error: {e}",
                'data': None}
```

This is just a simple example, but you can imagine how it will look like in a real application. And even if you handle this by yourself, you will have a frontend development cycle, with no type-hinting, no auto-completion again.

You can use `pywebchannel` library, and create your action like this:

```python
from pywebchannel import Action, Notify


# Inside your controller class
@Action(Notify([Todo]))
def addTodo(self, todo: Todo):
    # Do something with todo
    return todo
```

All the problems mentioned above are solved by `pywebchannel` library's `@Action` decorator. You can focus on your actual project .

2.2.5 Okay we almost there

Imagine that you have created your `Signals`, `Propertys` and `Slots` etc, in your controller class. But your frontend still does not recognize your backend types. You have to define all the types in your frontend typescript definition files, and keep them updated with your backend types. This is a nightmare, and you will have many bugs, and emotional-damages .

Fortunately `pywebchannel` library has a solution for this. You can use pywebchannel library's `ts_generator` tool. This is a simple script that can monitor your python files, and generate typescript definition files automatically. When you run it in a separate terminal, it will do its magic, and you will have a wonderful development experience .

To use this tool, you have to inherit your controller class from `pywebchannel.Controller` class, and use `pywebchannel` library's `Signal`, `Property` and `Action` instead of the ones provided by Qt. Also, you have to use `pydantic` for your model classes, which is the usual case for model classes in any project. That's it.

Please check the API documentation and example projects for more details.

2.3 Installation

You can simply install the package using pip:

```
pip install pywebchannel
```

All requirements will be installed automatically.

Requirements:

- PySide6 - Qt for Python
- pydantic - Model definition and validation
- colorama - Colored terminal output

2.4 Usage Setup :

2.4.1 Definition 1: Business Logic / A python project / Backend

Whatever you call for this step, it is just a regular Qt (PySide6) powered python project, which can take the advantage of full power of python with no limitation. To simplify the discussion here, it uses web socket(s) for real time communication and exposes objects through web socket(s). The `properties`, `methods`, `signal/notifiers` immediately become available to the UI with complete signature and type checking through type-script interfaces. Don't worry about complicated processes for managing sockets, it is not your responsibility. This is handled automatically, you can simply focus on your project.

2.4.2 Definition 2: User Interface / A web project / Frontend

Similarly, it is just a regular web project, which exploit the available modern UI tools. There is no limitation such as magically manipulated `window` interfaces or any complicated middle-ware translator which limits the functionality web library of yours.

2.4.3 `Step 1`: Create a meaningful directory structure.

It is completely up to you. But having a meaningful directory structure could make things simpler. For that purpose suggested way is to create a root directory with your `AppName` and two folders under the root directory, `backend` and `frontend`. As names suggest, they will be holding your python project as `backend` and UI project as `frontend`.

```
AppName
  backend
    ...
  frontend
    ...
  README.md
  LICENSE
  .gitignore
```

`Optinal virtual environment`: If you prefer using virtual environment for your python projects (which is the suggested way for any python project), create one virtual environment, and use it under your `backend` folder.

2.4.4 `Step 2`: Install the library `pywebchannel`

```
pip install pywebchannel
```

2.4.5 `Step 3`: Create an entry point for your `backend`.

Add a main file with any name, i.e. `main.py`, inside your `backend` folder. The entry point will contain python `main` function and will create a `QApplication` and run it. The responsibility of the entry point is to initiate the `WebChannelService` object(s) (Yes, you read it right, it is plural, you can create more than one communication channels to your UI application, for different purposes). Addition to that `main` needs to create the object(s) (at least the ones which need to be available at the beginning), and register those object(s) to the related `WebChannelService`.

```python
# main.py
import sys
from PySide6.QtWidgets import QApplication

from pywebchannel import WebChannelService

if __name__ == "__main__":
    app = QApplication(sys.argv)

    # Create a WebChannelService with a desired serviceName and the parent QObject
    commandTransferService = WebChannelService("Command Transfer Service", app)
```

(continues on next page)

(continued from previous page)

```
    # Start the service with a desired port number, 9000 in this example
    commandTransferService.start(9000)

    ...
    ...
    ...

    app.exec()
```

2.4.6 `Step 4`: Create a python package

To hold classes which contains functionalities to be invoked from `frontend`. Typically, it is better to create two packages, one for functionality classes and one for fixed structured objects, even though the second one is optional, it is completely okay to create it, no harm will be done if it is empty. The names of these folders could be anything, but having meaningful names would be helpful. Let's call them `controllers`, `models` respectively.

```
AppName
  backend
    controllers
      __init__.py
    models
      __init__.py
  frontend
    ...
  main.py
  README.md
  LICENSE
  .gitignore
```

2.4.7 `Step 5`: Create a controller class under your `controllers` package.

This is going to be one of the Type you are going to expose to your UI. Let's call it `HelloWorldController`. And make this class derived from `Controller`, which is imported from `pywebchannel`. Then, in your `main`, create an instance of it and register it into the `WebChannelService`.

```python
# controllers/HelloWorldController.py
from typing import Optional
from PySide6.QtCore import QObject
from pywebchannel import Controller


# Create a Controller class
class HelloWorldController(Controller):
    def __init__(self, parent: Optional[QObject] = None):
        # Controller name is typically the name of the class '__name__' attribute could be used as well
        super().__init__("HelloWorldController", parent)
```

And in main:

```python
# main.py
import sys
from PySide6.QtWidgets import QApplication
from pywebchannel import WebChannelService
from controllers.HelloWorldController import HelloWorldController

if __name__ == "__main__":
    app = QApplication(sys.argv)
    commandTransferService = WebChannelService("Command Transfer Service", app)
    commandTransferService.start(9000)

    # Create hello world controller object
    hwController = HelloWorldController(app)
    # Register controller for the communication service
    commandTransferService.registerController(hwController)

    app.exec()
```

2.4.8 `Step 6:` Add functionality

Technically, at this point our object, `hwController` has been already exposed to the any target UI. The functionality and properties of it is already accessible through a websocket located at port number 9000. The problem is that there is no functionality in our controller yet. Let's add a method into our controller, and decorate this method with a decorator named `Action` imported from `pywebchannel`

```python
# controllers/HelloWorldController.py
from typing import Optional
from PySide6.QtCore import QObject
from pywebchannel import Controller, Action


class HelloWorldController(Controller):
    def __init__(self, parent: Optional[QObject] = None):
        super().__init__("HelloWorldController", parent)

    # Create a class method and decorate it with @Action() decorator.
    # Don't forget to put annotations in your arguments. It is important!
    @Action()
    def sayHello(self, name: str):
        return f"Hello from 'HelloWorldController.sayHello' to my friend {name}"
```

2.4.9 Step 7: Create UI project

Now, we can try to use this inside a web app. For simplicity, inside the frontend, just create a Vite project with vanilla typescript template. You can create it yourself easily, or you can take it from examples folder.

2.4.10 Step 8: Establish connection

To establish connection between your backend and frontend, it is necessary to open a websocket connection from frontend to backend. Luckily, we can use built-in WebSocket in our frontend project. First create an api folder under your src and qwebchannel under that. Then populate the folder with given helpers in the repository examples (Just copy and paste the content into your project). Addition to those, you can create controllers and models directories as well, for nicely formatted structure.

```
AppName
  backend
    controllers
      __init__.py
      HelloWorldController.py
    models
      __init__.py
  frontend
    node_modules
    public
    src
      api
        controllers
        models
        qwebchannel
         index.d.ts
         index.js
     main.ts
     vite-env-d.ts
   index.html
   package.json
   tsconfig.json
   .gitignore
  main.py
  README.md
  LICENSE
  .gitignore
```

2.4.11 `Step 9`: Add QWebChannel javascript interface

`api/qwebchannel/index.js` is the official QWebChannel javascript interface. However, it is different than the original (`index_org.js`) one. It has been updated to support `async/await` pattern instead of old-school callback style usage. Addition to that a typescript definition has been attached as well, `index.d.ts`.

2.4.12 `Step 10`: Websocket Helper

Now, create a class to handle the websocket communication boiler-plate. You can use given `BaseAPI.ts` and `CommandAPI.ts` from the repository. The important part here is the implementation of `onChannelReady` callback located under `CommandAPI.ts`. This is the part where you access your object exposed from `backend` . As you guess, this access will be storing the reference to that object inside our API object, so that we can use it whenever we need it. Update the `CommandAPI.ts` for learning purposed debugging

```
// Inside CommadAPI.ts copied from repository
export class CommandAPI extends BaseAPI {
  public constructor() {
    super("ws://localhost:9000", "Command Transfer Service");
  }

  // Update this part to see the channel content.
  async onChannelReady(channel: QWebChannel): Promise<void> {
    console.log(channel)
  }
}
```

Then add connection request into your `main.ts`

```
// main.ts
// import API
import {API} from "./api/CommandAPI.ts";

// Try to connect
API.connect().then(() => {
  if (API.isConnected()) {
    console.log("Successfully connected to backend")
  }
}).catch((error) => {
  console.log(error)
})

// Add a simple UI
document.querySelector<HTMLDivElement>('#app')!.innerHTML = `
  <div>
    <input id="input">
    <button id="button">Say Hi</button>
  </div>
`

const input = document.querySelector<HTMLInputElement>('#input');
const button = document.querySelector<HTMLButtonElement>('#button');
```

(continues on next page)

(continued from previous page)

```
button?.addEventListener('click', () => {
  console.log(input?.value)
})
```

2.4.13 Step 11: Run the projects

Now, run the backend project, and run the frontend project. If everything is correct, you should see an output from backend terminal:

```
[INFO] - Command Transfer Service: 'Command Transfer Service' is active at PORT=9000
[INFO] - Command Transfer Service: New Connection (Active client count: 1)
```

and something similar from frontend browser console.

```
QWebChannel {...}
Successfully connected to backend
```

When you expand the QWebChannel object on the console, you should see an objects and HelloWorldController inside it. If this is the case, you have successfully connected your python backend to your frontend

2.4.14 Step 12: Use it

Let's use it and let our backend say hello to frontend. Just update your code as it should be:

Update CommandAPI.ts

```
export class CommandAPI extends BaseAPI {
  // Add an attribute for our API object
  HelloWorldController!: any;

  public constructor() {
    super("ws://localhost:9000", "Command Transfer Service");
  }

  async onChannelReady(channel: QWebChannel): Promise<void> {
    // Initialize it by the object located inside the QWebChannel
    this.HelloWorldController = channel.objects.HelloWorldController;
  }
}
```

Update main.ts

```
button?.addEventListener('click', async () => {
  // Call say hello with input value taken from input text box
  const response = await API.HelloWorldController.sayHello(input?.value)
  if (response.error) {
    // if an error occurred, display it
    console.log(response.error)
```

(continues on next page)

(continued from previous page)

```
    return
  }

  if (response.success) {
    // if a success message has been received, display it
    alert(response.success)
    return
  }

  if (response.data) {
    // if an extra data has been received, display it
    alert(JSON.stringify(response.data))
    return
  }
})
```

Refresh your web page, and click the button. You should see an alert message saying Hello from 'HelloWorldController.sayHello' to my friend <your input value>

2.4.15 Step 13: Type hinting

Everything is ready to go. The ONLY MISSING part is type hint in our frontend, because we don't have any type defintion for our controller, HelloWorldController. Type script generator given by pywebchannel comes in play at this moment.

- Step 1: Copy ts_generator.py and Paste it into your backend root folder, same level with your main.py.
- Step 2: Check the folder paths written in ts_generator.py script
- Step 3: Run it in separate terminal.

```
python ts_generator.py
```

- Step 4: You will see that the frontend controller folder will be populated with an auto generated Type-script interface, api/controllers/HelloWorldController.ts
- Step 5: Since it needs to use Response interface for type-hinting for return values, it needs to be located inside api/models directory, which is not there yet. Please copy it from the repoository. And also copy the Signal interface as well, which is going to be necessary when you use signals.
- Step 6: Now return back to your button.click listener implementation in main.ts. You will see that the function, sayHello(...), the return value response all are taking advantage of type-hinting.

2.4.16 `The last step`: Finalize your UI and serve it

When you complete your UI, you can serve it inside your `backend` project. For that purpose, you have a couple of options as usual.

First of all, you need to build your UI project. Inside your UI project, run:

```
npm run build
```

This is going to create a `dist` folder inside your UI project, `frontend/dist`. Copy `dist` folder into your `backend` project, and rename it with a meaningful name, such as `app_ui`.

Since it is a `javascript` based web project, opening the html file is not enough. The `javascript` functionalities will not be available. For that purpose, you need to serve it through a web server.

You can use the given `HttpServer` class inside `pywebchannel` for that purpose. It is a simple `http` server, which serves the given folder. Then you can access your UI through a browser, or even better, you can use `QWebEngineView` to display it

Please update your `main.py` file as follows. Feel free to use all the features you deserve from `QWebEngineView`:

```python
# main.py
import sys

from PySide6.QtCore import QUrl
from PySide6.QtWebEngineCore import QWebEngineSettings
from PySide6.QtWebEngineWidgets import QWebEngineView
from PySide6.QtWidgets import QApplication

from controllers.HelloWorldController import HelloWorldController
from pywebchannel import WebChannelService, HttpServer

if __name__ == "__main__":
    app = QApplication(sys.argv)

    # Create a WebChannelService with a desired serviceName and the parent QObject
    commandTransferService = WebChannelService("Command Transfer Service", app)
    # Start the service with a desired port number, 9000 in this example
    commandTransferService.start(9000)

    # Create hello world controller object
    hwController = HelloWorldController(app)
    # Register controller for the communication service
    commandTransferService.registerController(hwController)

    # Create http server and start it
    UI_PORT = 12000
    httpServer = HttpServer("app_ui", UI_PORT, app)
    httpServer.start()

    # Website on QTGui
    view = QWebEngineView()
    view.settings().setAttribute(QWebEngineSettings.WebAttribute.PluginsEnabled, True)
    view.settings().setAttribute(QWebEngineSettings.WebAttribute.DnsPrefetchEnabled,␣
↪True)
    view.load(QUrl(f"http://localhost:{UI_PORT}/"))
```

(continues on next page)

(continued from previous page)

```
    view.setWindowTitle("Hello World App")
    view.show()

    app.exec()
```

This will spin a web server at port `12000`, and serve the `app_ui` folder, and creates a `QWebEngineView` to display your UI. You can also access your UI through a browser by typing `http://localhost:12000` into the address bar. This could be helpful, if you observe any weird behaviour on your UI. The console on the browser could be helpful to debug the problem.

2.5 Congratulations!!!

You've successfully linked your Python backend to the frontend, introducing a tool capable of dynamically generating TypeScript interfaces by monitoring backend changes. This tool ensures that your frontend remains synchronized with backend updates, streamlining the development process.

As long as the tool is active, it automatically updates scripts as you modify the backend, maintaining consistency between the two worlds. Now, you can explore additional examples and delve into the self-explanatory API.

Consider the following steps as you continue to enhance your development process:

1. Documentation and Usage Guidelines:

- Develop comprehensive documentation to guide users on effectively utilizing the tool.
- Provide clear instructions on structuring the backend code to optimize the automatic generation of TypeScript interfaces.

2. Expand Script Generation:

- Explore opportunities to extend the tool's capabilities beyond TypeScript interfaces, such as generating API documentation or other relevant artifacts based on backend modifications.

3. User Interface for the Tool:

- Enhance accessibility by creating a user interface for the tool, catering to developers who may prefer graphical interfaces over command-line tools.
- Implement features like configurable options and settings to further customize the tool's behavior.

2.6 How to Contribute

If you want to contribute to this project, you are more than welcome. Here are some ways you can help:

- Report any bugs or issues you find.
- Suggest new features or improvements.
- Submit pull requests with your code changes.
- Share your feedback or suggestions.

2.7 License

This project is licensed under the MIT License. See the LICENSE file for details.

2.8 Credits

This project was inspired by the following sources:

- QWebChannel - a Qt module that enables seamless integration of C++ and HTML/JavaScript.
- PySide6 - a Python binding of the cross-platform GUI toolkit Qt.
- TypeScript - a superset of JavaScript that adds optional types.

2.9 API

2.9.1 Controller

`pywebchannel.Controller.Action`(*notify: Notify = None*)

A decorator that converts a Python function into a Qt slot. The notify argument is used to emit after the function is executed. Defaults to None. If it is specified, a signal with the given name will be created and attached into the class. You don't need to create that signal yourself. The signal will be emitted with the result of the function. EmitBy is used to specify the source of the notification. If it is set to EmitBy.Auto, the notification will be emitted automatically after the function is executed. If it is set to EmitBy.User, the notification will be emitted only if the function explicitly emits it.

Parameters

notify (`Notify`, *optional*) – A Notify object that specifies the name and arguments of a notification signal

Returns

A wrapper function that is a Qt slot with the same arguments and return type as the original function. The slot also handles serialization and deserialization of inputs and outputs, exception handling,and optionally emits a notification signal with the result.

References

https://doc.qt.io/qtforpython-6/tutorials/basictutorial/signals_and_slots.html

See also:

Signal, Property

`class pywebchannel.Controller.Controller`(*controllerName: str, parent: QObject | None = None*)

Bases: `QObject`

A base class for controllers that provides common functionality.

`_controllerName`

A private instance attribute that stores the name of the controller.

cleanup() → None

Performs any necessary cleanup actions before the controller is destroyed.

This method can be overridden by subclasses to implement their own cleanup logic.

name() → str

Returns the name of the controller.

Returns

A string that represents the name of the controller.

staticMetaObject = PySide6.QtCore.QMetaObject("Controller" inherits "QObject":)

class pywebchannel.Controller.**Convert**

Bases: `object`

This class provides some utility methods to convert data types between Python, Qt, and web formats.

static **from_py_to_qt**(*argDict: Dict[str, type]*) → Tuple[List[str], List[type]]

Converts a dictionary of argument names and types from Python to Qt format.

- Primitive types are kept as they are.
- List types are converted to list type.
- Pydantic types are converted to dict type.
- Other types are converted to dict type.

Returns

Tuple[List[str], List[type]] - argument names and argument types.

static **from_py_to_web**(*arg*) → Any

Converts a Python format argument to a web format argument.

Returns

- Primitive types are kept as they are.
- List types are recursively converted using the inner type.
- Pydantic types are converted to a dictionary using the dict() method.
- Other types are converted to a dictionary using the dict() method.

static **from_py_to_web_response**(*result*) → Dict[str, Any]

Converts a Python format result to a web format response.

- String types are wrapped in a Response object with success attribute.
- Response types are converted to a dictionary using the dict() method.
- Other types are wrapped in a Response object with data attribute.

Returns

Dict[str, Any] - a dictionary that represents the response.

static **from_web_to_py**(*arg*, *paramType*) → Any

Converts a web format argument to a Python format argument according to the given parameter type.

Returns

- Primitive types are kept as they are.
- List types are recursively converted using the inner type.
- Pydantic types are instantiated using the argument as a keyword dictionary.
- Other types are kept as they are.

`class pywebchannel.Controller.`**`EmitBy`**

Bases: `object`

A class to represent the source of a notification.

`Auto = 0`

`User = 1`

`class pywebchannel.Controller.`**`Helper`**

Bases: `object`

`static infer_caller_info`(*stack: List[FrameInfo]*) → Tuple[str, str]

A method that infers the name of the controller and the variable that called this method from the stack trace.

Parameters

`stack` (*`List[inspect.FrameInfo]`*) – A list of frame information objects representing the current call stack.

Returns

A tuple of two strings: the name of the controller and the name of the variable that called this method. If the variable name cannot be inferred, an empty string is returned as the second element of the tuple.

Return type

Tuple[str, str]

`class pywebchannel.Controller.`**`Notify`**(*arguments: Dict[str, type] | List[type], name: str = None, emitBy: EmitBy = 0*)

Bases: `object`

A class to represent a notification object.

`name`

The name of the notification.

Type

str

`arguments`

A dictionary of the arguments that the notification expects, with the argument

Type

Dict[str, type]

`name as the key and the argument type as the value.`

`emitBy`

The source of the notification, either EmitBy.Auto or EmitBy.User.

Type

EmitBy

`The default value is EmitBy.Auto.`

`pywebchannel.Controller.`**`Property`**`(`*p_type: type*, *init_val=None*, *get_f=None*, *set_f=None*`)` → Property

A function that creates a Qt property and a corresponding signal. The function is responsible for creating the backend variable, getter and setter functions, and the signal object related with the property.

Parameters

- **`p_type`** (*`type`*) – The type of the property value.
- **`init_val`** – The initial value of the property. Defaults to None
- **`get_f`** (*`function, optional`*) – A custom getter function for the property. Defaults to None.
- **`set_f`** (*`function, optional`*) – A custom setter function for the property. Defaults to None.

Returns

The prop which is a QtCore.Property object.

Raises

`Exception` – If the property name cannot be inferred from the caller information

References

https://doc.qt.io/qtforpython-6/PySide6/QtCore/Property.html

See also:

Signal, Action

`class pywebchannel.Controller.`**`Response`**`(`*, *success: str* | *None = None*, *error: str* | *None = None*, *data: Any* | *None = None*`)`

Bases: `BaseModel`

A Pydantic model that represents the outcome of some operation.

`data: Any | None`

Any Python object that stores the result of the operation. It can be of any type, such as a dict, a list, a tuple, a string, a number, etc. Pydantic will not perform any validation or conversion on this field.

`error: str | None`

A string that provides an error message if something went wrong during the operation. It can be None or any string value. For example, “Invalid input”, “Connection timeout”, “Database error”etc.

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Union[Any, NoneType], required=False), 'error': FieldInfo(annotation=Union[str, NoneType], required=False), 'success': FieldInfo(annotation=Union[str, NoneType], required=False)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

`success: str | None`

A string that indicates whether the operation was successful or not. It can be None or any string value. For example, “yes”, “no”, “ok”, “error”, etc.

`pywebchannel.Controller.`**`Signal`**`(`*args: Dict[str, type] | List[type], controllerName: str = None, signalName: str = None*`)` → Signal

A function that creates a Qt signal with the given arguments by making necessary type conversions to keep Qt and serialization process happy.

Parameters

- **`args`** (*`Dict[str, type] or List[type]`*) – A dictionary that maps the names and types of the signal arguments.
- **`controllerName`** (*`str, optional`*) – The name of the controller that defines the signal. Defaults to None.
- **`signalName`** (*`str, optional`*) – The name of the signal. Defaults to None.

Returns

A QtCore.Signal object with the specified arguments, name, and arguments names.

Raises

`Exception` – If the controller name or signal name cannot be inferred from the caller information, or if the signal name is empty.

References

https://doc.qt.io/qtforpython-6/PySide6/QtCore/Signal.html

See also:

Property, Action

`class pywebchannel.Controller.`**`Type`**

Bases: `object`

This class provides some utility methods to check the type of variable.

`is_primitive(var_type`

type) -> bool: Returns True if the given type is a primitive type, False otherwise.

`is_list(var_type`

type) -> bool: Returns True if the given type is a list type, False otherwise.

`is_pydantic(var_type`

type) -> bool: Returns True if the given type is a subclass of pydantic.BaseModel, False otherwise.

`static `**`is_list`**`(`*var_type: type*`)`

`static `**`is_primitive`**`(`*var_type: type*`)`

`static `**`is_pydantic`**`(`*var_type: type*`)`

`primitives = (<class 'bool'>, <class 'str'>, <class 'int'>, <class 'float'>, <class 'NoneType'>)`

A tuple of primitive types in Python, such as bool, str, int, float, and NoneType.

2.9.2 GeneratorWatcher

class pywebchannel.GeneratorWatcher.**GeneratorWatcher**(*parent: QObject* | *None* = *None*)

Bases: `QFileSystemWatcher`

A class that inherits from QFileSystemWatcher and watches for changes in python files.

watchTargetDirMap

A dictionary that maps the source directory to the target directory.

Type

Dict[str, str]

addDirectory(*dirPathToWatch: str*, *dirTargetPath: str*)

A method that adds a directory to the watch list.

Parameters

- **dirPathToWatch** (*str*) – The path of the directory to watch.
- **dirTargetPath** (*str*) – The path of the target directory to generate typescript files.

addFile(*filePath: str*)

A method that adds a file to the watch list.

Parameters

filePath (*str*) – The path of the file to watch.

getOutputFilePath(*filePath*)

Get the output file path for the TypeScript file.

Parameters

filePath (*str*) – The input file path for the Python file.

Returns

The output file path for the TypeScript file.

Return type

str

onDirectoryChanged(*dirPath: str*)

The slot that is triggered when a directory is changed.

Parameters

dirPath (*str*) – The path of the changed directory.

onFileChanged(*filePath: str*)

The slot that is triggered when a file is changed.

Parameters

filePath (*str*) – The path of the changed file.

staticMetaObject = PySide6.QtCore.QMetaObject("GeneratorWatcher" inherits "QFileSystemWatcher": Methods: #9 type=Slot, signature=onDirectoryChanged(QString), parameters=QString #10 type=Slot, signature=onFileChanged(QString), parameters=QString)

2.9.3 WebChannelService

class pywebchannel.WebChannelService.**WebChannelService**(*serviceName: str*, *parent: QObject | None = None*)

Bases: `QObject`

A class that inherits from QObject and provides a web channel service using QWebSocketServer and QWebChannel.

websocketServer

The QWebSocketServer object that provides the WebSocket server.

Type

QWebSocketServer

port

The port number for the WebSocket server.

Type

int

serviceName

The name of the web channel service.

Type

str

clientWrapper

The WebSocketClientWrapper object that handles the WebSocket connections from the server.

Type

WebSocketClientWrapper

channel

The QWebChannel object that manages the communication between the server and the clients.

Type

QWebChannel

activeClientCount

The number of active WebSocket clients connected to the server.

Type

int

isOnline() → bool

Checks if the web channel service is online by checking the status of the WebSocket server.

Returns

True if the web channel service is online, False otherwise.

Return type

bool

onClientConnected(*transport: WebSocketTransport*) → None

Connects the web channel to the WebSocket transport and increments the active client count.

This slot is invoked when the clientWrapper object emits the clientConnected signal.

Parameters
`transport` (`WebSocketTransport`) – The WebSocketTransport object that represents the WebSocket connection.

`onClientDisconnected`(*transport:* WebSocketTransport) → None

Decrements the active client count and cleans up the controller objects if the active client count is zero.

This slot is invoked when the clientWrapper object emits the clientDisconnected signal.

Parameters
`transport` (`WebSocketTransport`) – The WebSocketTransport object that represents the WebSocket connection.

`onClosed`() → None

Logs the information of closing the web channel service.

This slot is invoked when the websocketServer object emits the closed signal.

`registerController`(*controller:* Controller) → None

Registers a controller object to the web channel using the channel attribute.

Parameters
`controller` (`Controller`) – The controller object to be registered.

`start`(*port: int*) → bool

Starts the web channel service by creating and listening to a WebSocket server at the given port.

Parameters
`port` (*`int`*) – The port number for the WebSocket server.

Returns
True if the web channel service is started successfully, False otherwise.

Return type
bool

`staticMetaObject = PySide6.QtCore.QMetaObject("WebChannelService" inherits "QObject": Methods: #5 type=Slot, signature=onClosed() #6 type=Slot, signature=onClientConnected(QWebChannelAbstractTransport*), parameters=QWebChannelAbstractTransport* #7 type=Slot, signature=onClientDisconnected(QWebChannelAbstractTransport*), parameters=QWebChannelAbstractTransport*)`

`stop`() → None

Stops the web channel service by closing and deleting the WebSocket server.

class `pywebchannel.WebChannelService.`**`WebSocketClientWrapper`**(*server: QWebSocketServer*, *parent: QObject* | None = *None*)

Bases: `QObject`

A class that inherits from QObject and handles the WebSocket connections from a QWebSocketServer.

`server`

The QWebSocketServer object that listens for WebSocket connections.

Type
QWebSocketServer

`clientConnected`

The signal that is emitted when a new WebSocket connection is established.

clientDisconnected

The signal that is emitted when an existing WebSocket connection is closed.

handleNewConnection() → None

Creates a WebSocketTransport object for the next pending connection from the server and emits the clientConnected signal.

This slot is invoked when the server object emits the newConnection signal.

staticMetaObject = PySide6.QtCore.QMetaObject("WebSocketClientWrapper" inherits "QObject": Methods: #5 type=Signal, signature=clientConnected(QWebChannelAbstractTransport*), parameters=QWebChannelAbstractTransport* #6 type=Signal, signature=clientDisconnected(QWebChannelAbstractTransport*), parameters=QWebChannelAbstractTransport* #7 type=Slot, signature=handleNewConnection())

class pywebchannel.WebChannelService.**WebSocketTransport**(*socket: QWebSocket*)

Bases: `QWebChannelAbstractTransport`

A class that inherits from QWebChannelAbstractTransport and communicates with a QWebSocket.

socket

The QWebSocket object that handles the WebSocket connection.

Type
QWebSocket

disconnected

The signal that is emitted when the socket is disconnected.

onSocketDisconnected() → None

Emits the disconnected signal with the self object and deletes the self object and the socket object.

This slot is invoked when the socket object emits the disconnected signal.

sendMessage(*message*) → None

Sends a message to the WebSocket using the socket object.

The message is converted to a QJsonDocument and then to a compact JSON string.

Parameters
message – The message to be sent.

staticMetaObject = PySide6.QtCore.QMetaObject("WebSocketTransport" inherits "QWebChannelAbstractTransport": Methods: #7 type=Signal, signature=disconnected(QWebChannelAbstractTransport*), parameters=QWebChannelAbstractTransport* #8 type=Slot, signature=onSocketDisconnected() #9 type=Slot, signature=textMessageReceived(QString), parameters=QString)

textMessageReceived(*messageData: str*) → None

Receives a text message from the WebSocket using the socket object and emits the messageReceived signal.

The text message is parsed as a QJsonDocument and then as a QJsonObject. If there is any error in parsing, the error is logged using the Logger object.

This slot is invoked when the socket object emits the textMessageReceived signal.

Parameters
messageData (*str*) – The text message received from the WebSocket.

2.9.4 HttpServer

class pywebchannel.HttpServer.**HttpServer**(*serverDir: str*, *port: int*, *parent=None*)

Bases: `QObject`

A class that inherits from QObject and runs an HTTP server using QProcess.

process

The QProcess object that executes the HTTP server.

Type

QProcess

port

The port number for the HTTP server.

Type

int

serverDir

The directory path for the HTTP server.

Type

str

onReadyReadStandardError() → None

Reads the standard error from the QProcess object and logs it using the Logger object.

This slot is invoked when the QProcess object emits the readyReadStandardError signal.

onReadyReadStandardOutput() → None

Reads the standard output from the QProcess object and logs it using the Logger object.

This slot is invoked when the QProcess object emits the readyReadStandardOutput signal.

start() → None

Starts the HTTP server using the QProcess object.

The QProcess object executes the command "python -m http.server port –directory serverDir".

staticMetaObject = PySide6.QtCore.QMetaObject("HttpServer" inherits "QObject": Methods: #5 type=Slot, signature=stop() #6 type=Slot, signature=onReadyReadStandardOutput() #7 type=Slot, signature=onReadyReadStandardError())

stop() → None

Stops the HTTP server by killing the QProcess object.

Logs the information of stopping the HTTP server using the Logger object.

2.9.5 CodeAnalyzer

`class pywebchannel.CodeAnalyzer.CodeAnalyzer(`*MetaClass*`)`

Bases: `object`

A class that analyzes the code of a given class and determines its type and acceptability.

`MetaClass`

The class object to be analyzed.

Type

type

`_classType`

The type of the class object, one of the supported types.

Type

str

`_isAcceptable`

A flag indicating whether the class object is acceptable for analysis or not.

Type

bool

`classType()`

Returns the type of the class object, one of the supported types.

Returns

The type of the class object, or an empty string if not acceptable.

Return type

str

`isAcceptable()`

Returns whether the class object is acceptable for analysis or not.

Returns

True if the class object is acceptable, False otherwise.

Return type

bool

`run()`

Runs the analysis on the class object and returns an interface object.

Returns

The interface object corresponding to the class object's type, or None if not acceptable.

`class pywebchannel.CodeAnalyzer.ControllerInterface(`*MetaClass*`)`

Bases: *`Interface`*

A class that represents the interface of a controller class.

A controller class is a subclass of QObject that defines properties, signals, and slots that can be used to communicate with other classes or components.

`props`

The properties of the controller class.

Type

list of *Property*

`signals`

The signals of the controller class.

Type
list of *Signal*

`slots`

The slots of the controller class.

Type
list of *Slot*

`classType()`

Returns the type of the interface, which is SupportedTypes.Controller.

Returns
The type of the interface.

Return type
str

`dependencies()`

Returns the list of dependencies of the interface.

Dependencies are the types that are used by the properties, signals, and slots of the interface.

Returns
The list of dependencies, without duplicates.

Return type
list[str]

`class pywebchannel.CodeAnalyzer.Interface(`*MetaClass*`)`

Bases: `object`

A base class that represents the interface of a class.

An interface is a set of properties, signals, and slots that define the communication and functionality of a class.

`MetaClass`

The metaclass of the class that implements the interface.

Type
type

`name`

The name of the interface.

Type
str

`objectDict`

The dictionary of the meta class's attributes and methods.

Type
dict

`staticMetaObject`

The static meta-object of the meta class.

Type
QMetaObject

props

The properties of the interface.

Type

list of *Property*

signals

The signals of the interface.

Type

list of *Signal*

slots

The slots of the interface.

Type

list of *Slot*

classType()

Returns the type of the interface, which is “Interface”.

Returns

The type of the interface.

Return type

str

dependencies()

Returns the list of dependencies of the interface.

Dependencies are the types that are used by the properties, signals, and slots of the interface.

Returns

The list of dependencies, without duplicates.

`class pywebchannel.CodeAnalyzer.`**`ModelInterface`**`(`*MetaClass*`)`

Bases: `Interface`

A class that represents the interface of a model class.

props

The properties of the model class.

Type

list of *Property*

classType()

Returns the type of the interface, which is SupportedTypes.Model.

Returns

The type of the interface.

Return type

str

dependencies()

Returns the list of dependencies of the interface.

Dependencies are the types that are used by the properties of the interface.

Returns

The list of dependencies, without duplicates.

Return type
list[str]

`class pywebchannel.CodeAnalyzer.`**`Parameter`**`(`*name: str*, *typeStr: str*`)`

Bases: `object`

A class to represent a parameter in TypeScript.

`name`

The name of the parameter.

Type
str

`type`

The type of the parameter in TypeScript syntax.

Type
str

`code`

The code representation of the parameter.

Type
str

`convertCode``()` → None

Generate the code representation of the parameter.

`convertType``()` → None

Convert the type attribute to a TypeScript compatible type.

`dependencies``()` → list[str]

Return a list of the dependencies of the parameter type.

Returns
A list of the types that the parameter type depends on, without brackets.

Return type
list[str]

`class pywebchannel.CodeAnalyzer.`**`Property`**`(`*name: str*, *typeStr: str*`)`

Bases: `object`

`convertCode``()` → None

Generate the code attribute for the property.

`convertType``()` → None

Convert the type attribute to a TypeScript compatible type.

`dependencies``()`

Return the dependencies of the property.

Returns
A list of the types that the property depends on.

Return type
list

`class pywebchannel.CodeAnalyzer.`**`Return`**`(`*typeStr: str*`)`

Bases: `object`

convertCode() → None

Generate the code attribute for the return type.

convertType() → None

Convert the type attribute to a TypeScript compatible type.

dependencies()

Return the dependencies of the return type.

Returns
A list of the types that the return type depends on.

Return type
list

class pywebchannel.CodeAnalyzer.**Signal**(*name: str*, *parameters: list[Parameter]*, *returnType: Return*)

Bases: object

convertCode() → None

Generate the code attribute for the signal.

convertType() → None

Convert the type attributes of the parameters and the returnType to TypeScript compatible types.

dependencies()

Return the dependencies of the signal.

Returns
A list of the types that the signal depends on.

Return type
list

class pywebchannel.CodeAnalyzer.**Slot**(*name: str*, *parameters: list[Parameter]*, *returnType: Return*)

Bases: object

A class to represent a slot of a TypeScript class.

name

The name of the slot.

Type
str

parameters

A list of Parameter objects that represent the parameters of the slot function.

Type
list[*Parameter*]

returnType

A Return object that represents the return type of the slot function.

Type
Return

code

The code for the slot declaration in TypeScript.

Type
str

convertCode() → None

Generate the code attribute for the slot.

convertType() → None

Convert the type attributes of the parameters and the returnType to TypeScript compatible types.

dependencies()

Return the dependencies of the slot.

Returns
A list of the types that the slot depends on.

Return type
list

class pywebchannel.CodeAnalyzer.**SupportedTypes**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `StrEnum`

Controller = 'Controller'

Model = 'BaseModel'

2.9.6 Utils

class pywebchannel.Utils.**Generator**

Bases: `object`

A class to generate TypeScript code for interfaces.

static **header**()

Generate the header for the TypeScript file.

Returns
A list of strings that represent the header lines.

Return type
list

static **imports**(*deps*)

Generate the import statements for the TypeScript file.

Parameters
deps (*list*) – A list of strings that represent the dependencies of the interfaces.

Returns
A list of strings that represent the import statements.

Return type
list

static **interface**(*name: str*, *interface*)

Generate the interface declaration for the TypeScript file.

Parameters

- **name** (*str*) – The name of the interface.
- **interface** (`Interface`) – An Interface object that represents the interface.

Returns
A list of strings that represent the interface declaration.

Return type
list

`class pywebchannel.Utils.Logger`

Bases: `object`

A class to log messages with different colors and levels.

`static error`(*message*, *sender=''*) → None

Log an error message with red color and optional sender name.

Parameters

- **message** (*str*) – The message to log.
- **sender** (*str*) – The name of the sender of the message. Default to "".

Returns
None

`static info`(*message*, *sender=''*) → None

Log an info message with green color and optional sender name.

Parameters

- **message** (*str*) – The message to log.
- **sender** (*str*) – The name of the sender of the message. Default to "".

Returns
None

`static status`(*message*, *sender=''*, *override=True*) → None

Log a status message with blue color and optional override flag.

Parameters

- **message** (*str*) – The message to log.
- **override** (*bool*) – A flag to indicate whether to override the previous status message or not. Default to True.

Returns
None

`static warning`(*message*, *sender=''*) → None

Log a warning message with yellow color and optional sender name.

Parameters

- **message** (*str*) – The message to log.
- **sender** (*str*) – The name of the sender of the message. Default to "".

Returns
None

`class pywebchannel.Utils.Utils`

Bases: `object`

A class that provides some utility methods for working with types and signatures.

VARIABLE_TYPE_MAP = {'QString': 'string', 'QVariantList': 'any[]', 'QVariantMap': 'any', 'Response': 'Response', 'any': 'any', 'bool': 'boolean', 'dict': 'any', 'double': 'number', 'float': 'number', 'int': 'number', 'list': 'any[]', 'str': 'string', 'void': 'void'}

static convertType(*text*) → str

Converts a Python type to a TypeScript type using the VARIABLE_TYPE_MAP.

Parameters

text – A string that represents the Python type to be converted.

Returns

A string that represents the TypeScript type, with the format '<type>[]' for list types.

static getInheritanceTree(*T: type*)

Returns a dictionary that represents the inheritance tree of a given type.

Parameters

T – A type object that represents the subclass.

Returns: A dictionary that maps the names of the base classes to their type objects, starting from the subclass to the object class.

static isList(*text: str*) → tuple[bool, str]

Checks if a string representation of a type is a list type.

Parameters

text – A string that represents the type to be checked.

Returns

A tuple of a boolean value and a string prefix. The boolean value is True if the type is a list type, and False otherwise. The string prefix is either list[or List[depending on the case of the type, or an empty string if the type is not a list type.

static isTypescriptPrimitive(*text: str*) → bool

Checks if a string representation of a type is a TypeScript primitive type.

Parameters

text – A string that represents the type to be checked.

Returns

A boolean value that is True if the type is a TypeScript primitive type, and False otherwise.

static parseWithInspect(*f*)

Parses the signature of a function using the inspect module.

Parameters

f (*function*) – The function to be parsed.

Returns

The names of the parameters of the function. paramTypes (list of str): The types of the parameters of the function, or empty strings if not annotated. returnType (str): The type of the return value of the function, or "Response" if not annotated.

Return type

paramNames (list of str)

pp = <pprint.PrettyPrinter object>

`static simplyVariableType`(*text: str*) → str

Simplifies a str representation of a type by removing whitespace, quotation marks, and package information.

Parameters

`text` – A string that represents the type to be simplified.

Returns

A simplified string that represents the type, with the format 'list[<type>]' for list types.

`static type_to_string`(*t: type*)

Returns a string representation of a given type.

Parameters

`t` – A type object that represents the type to be converted.

Returns

A string that represents the type, with the format 'list[<type>]' for list types.

PYTHON MODULE INDEX

INDEX