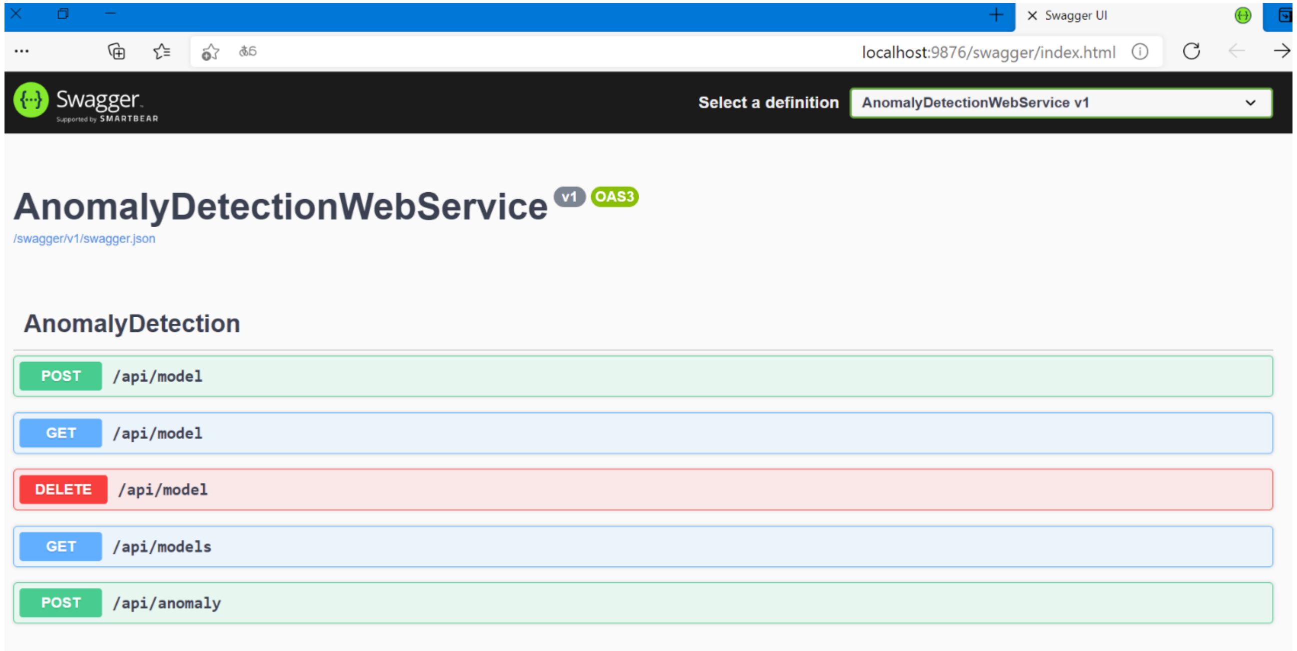


Swagger UI
localhost:9876/swagger/index.html
Swagger
Supported by SMARTBEAR
Select a definition
AnomalyDetectionWebService v1

AnomalyDetectionWebService v1 OAS3

/swagger/v1/swagger.json

AnomalyDetection

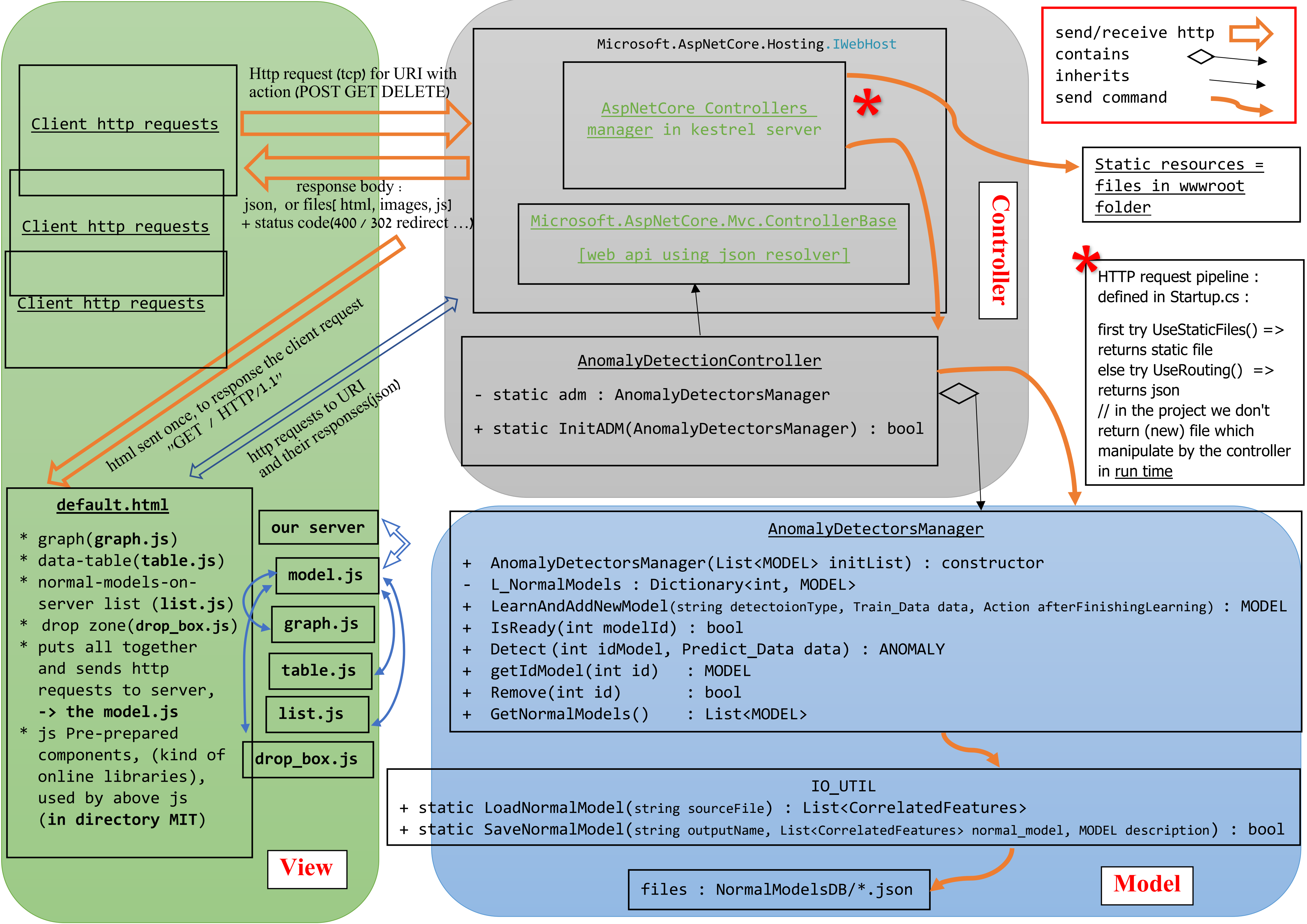
- POST /api/model
- GET /api/model
- DELETE /api/model
- GET /api/models
- POST /api/anomaly

Action + URI	Description	Client Request	Server Response	Client Request example* (A, B are features)	Server Response example* (A, B are features)
Post /api/model	Upload and create new normal model on server	**Query Parameter:** model_type="hybrid" | "regression" **Body:** {"train_data": **<DATA>**}	**<MODEL>**	**Query Parameter:** model_type="hybrid" **Body:** {"train_data": {“A”: [100, 100.203, 101 …], “B”: [-100, -100.203, -100.5 …], … }}	{"model_id":123, "upload_time": "2021-04-22T19:15:32+02.00", "status": "pending”}
Get /api/model	Get normal model status	**Query Parameter:** model_id=**<int>**	**<MODEL>**	**Query Parameter:** model_id=123	{"model_id":123, "upload_time": "2021-04-22T19:15:32+02.00", "status": "pending”}

Delete /api/model	Delete normal model from the server	**Query Parameter:** model_id=**<int>**	----	**Query Parameter:** model_id=123	----
Get /api/models	Get all normal model status	----	(**<MODEL>**)* = [<MODEL>, **<MODEL>** …]	----	[{"model_id":123, "upload_time": "2021-04-22T19:15:32+02.00", "status": "ready”}, …]
Post /api/anomaly	Find anomalies according to exists normal model	**Query Parameter:** model_id=**<int>** **Body:** {"predict_data": **<DATA> }**	**<ANOMALY>** = {"anomalies": {(**<string>**:(**<SPAN>**)*)* }, "reason": {(**<string>**:**<string>**)*}}	**Query Parameter:** model_id=123 **Body:** {"predict_data": **{“A”:** [100, 100.203, 101 …], “B”: [-1700, -1700.203, -1700.5 …], … **}**}	{"anomalies": **{**“A”: [[1, 40], [44, 120] …], “B”: [[1,40], [44, 120] …], …}, "reason": {“A”: "Linear regression with B", …}}
Get /	Get static server resource, a html page which includes JavaScript, for dynamic interaction in client-browser side. (A page that use the above URIs)				

*   Those are examples of the types that are sent over the http traffic,
For full information about what status http can be in the response, and data type,
see documentation in "AnomalyDetectionController.cs".

** We can say that **<MODEL>** = {"model_id": **<int>**, "upload_time": **<datetime>**, "status": "ready" | "pending"}
<DATA> = {(**<string>**: (**<float>**)*)* }   **<SPAN>** = [**<long>**, **<long>**]



send/receive http

contains

inherits

send command

Client http requests

Client http requests

Client http requests

Http request (tcp) for URI with action (POST GET DELETE)

response body :
json, or files[html, images, js]
+ status code(400 / 302 redirect …)

Microsoft.AspNetCore.Hosting.IWebHost

AspNetCore Controllers manager in kestrel server

*

Microsoft.AspNetCore.Mvc.ControllerBase

[web api using json resolver]

Controller

Static resources = files in wwwroot folder

*

HTTP request pipeline : defined in Startup.cs :

first try UseStaticFiles() => returns static file
else try UseRouting() => returns json
// in the project we don't return (new) file which manipulate by the controller in run time

html sent once, to response the client request
„GET / HTTP/1.1"

http requests to URI and their responses(json)

AnomalyDetectionController

- static adm : AnomalyDetectorsManager

+ static InitADM(AnomalyDetectorsManager) : bool

default.html

* graph(**graph.js**)
* data-table(**table.js**)
* normal-models-on-server list (**list.js**)
* drop zone(**drop_box.js**)
* puts all together and sends http requests to server, **-> the model.js**
* js Pre-prepared components, (kind of online libraries), used by above js (**in directory** MIT)

our server

model.js

graph.js

table.js

list.js

drop_box.js

View

AnomalyDetectorsManager

+ AnomalyDetectorsManager(List<MODEL> initList) : constructor
- L_NormalModels : Dictionary<int, MODEL>
+ LearnAndAddNewModel(string detectoionType, Train_Data data, Action afterFinishingLearning) : MODEL
+ IsReady(int modelId) : bool
+ Detect (int idModel, Predict_Data data) : ANOMALY
+ getIdModel(int id) : MODEL
+ Remove(int id) : bool
+ GetNormalModels() : List<MODEL>

IO_UTIL

+ static LoadNormalModel(string sourceFile) : List<CorrelatedFeatures>
+ static SaveNormalModel(string outputName, List<CorrelatedFeatures> normal_model, MODEL description) : bool

files : NormalModelsDB/*.json

Model

Raw short-text about the server-side classes:

- Microsoft.AspNetCore.Mvc.ControllerBase => for developing fast web controller which deals with URIs
- AnomalyDetectionController extents ControllerBase => handle HTTP request for few URIs
  privat static AnomalyDetectorsManager adm = null
  public static bool InitADM => to initialize adm member

- AnomalyDetectorsManager => used by AnomalyDetectionController to handle Mathematical/logical anomaly detection, but also in used for proper list management, thread safe.
  public LearnAndAddNewModel()
  public Detect()
  public getIdModel()
  public Remove()
  public GetNormalModels()
  Note that even if the exception is caught in debugging it might seems that it doesn't. [It is disadvantage so you can't run reliable server from the debugger]
  From : https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library
  When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue and see the exception-handling behavior that is demonstrated in these examples. To prevent Visual Studio from breaking on the first error, just uncheck the Enable Just My Code checkbox under Tools, Options, Debugging, General.

- Program => entry point
- Startup => helper class to config http server

Static classes:

- AnomalyDetection => used by AnomalyDetectorsManager to handle all Mathematical/logical aspect of anomaly detection
  public static GetNormal()
  public static GetDetection()
  public static ToSpanDictionary()
  public static GetReportTypes()

Utils:

- MathUtil => for AnomalyDetection class
- MinimalCircle => for MathUtil class
- IO_Util => for loading / store ExtendedModelInfo in json file, used by AnomalyDetectorsManager

```
public static LoadNormalModel() <=> IO files
public static RestoreExtendedModelInfo() <=> IO files
public static SaveNormalModel() <=> IO files
```

Classes which in use to define types:

- MODEL => same as <MODEL> above in this file, int model_id;DateTime upload_time; string status; all are public properties
- CorrelatedFeatures => describes two correlative features
- ExtendedModelInfo => has MODEL info AND List<CorrelatedFeatures> normal_model, meaning general info and the data of all correlation between features
- Predict_Data => has public property predict_data of Dictionary<String, List<float>>
- Train_Data => has public property train_data of Dictionary<String, List<float>>
- ANOMALY => has Dictionary<string, List<Span>> anomalies AND Dictionary<string, string> reason
- Point
- Line
- Circle
- Counter => wrapping class for int, in order lock(){} can be applied on it
- Span => alias for List<long> which should contain exactly 2 elements: [start(inclusive), end(exclusive)]

Raw short-text about the server and client-side:

From the whole project perspective, the server-response to the client, is the view.

However, the response can be json, or it can be file, and in this project, files that returned by the server are only static resources(from server perspective) meaning the server doesn't manipulate the page/icon that is returned to the client. Indeed default.html is dom + ajax file which will operate 'dynamic' in the browser.

So it can be described as MVC like that:

- **Server**:
    - **Model** :
        - IO (not separated to different layer in this project),
        - AnomalyDetectorsManger,
        - AnomalyDetection algorithm
    - **Controller**:
        - AnomalyDetectionController class
- **Client**:
    - **View**:
        - json responses or
        - static resources, for example:
            - favicon.ico, or
            - **default.html** (using ajax):
                - divided to 3 layers [user does action \ model.js refresh the data]:
                    - foreground layer (html, dom)
                    - ⇅
                    - middle layer (graph.js, table.js, list.js, drop_box.js)
                    - ⇅
                    - background layer [logic + connection with the server] (model.js)