Starting from this tutorial, you will explore how to create a Web API with a web framework.

1. You are NOT necessary to develop the API which connecting with a database at the beginning of the stage. If you are failed to create a “Hard Code” data API, you will not be able to read the data from the database.
2. The laboratory may use a framework that you are NOT familiar to, please be open mind to learn different Web Framework.
3. You are STRONGLY NOT ADVISE to use any framework that can help you to develop both frontend (with UI) and backend (API). Please be remind that you are NOT developing a server-side web application with web server support such as PHP or JSP or ASP.NET or you may misunderstand the concept of API development.
4. You are reminded to PUSH your code to your Git project once the complete the laboratory. Thus, **this is count as version control marks in your portfolio assignment**.

API Development

Koa is a progressive web framework for building efficient and scalable web applications. It is an open-source Node.js framework that is lightweight and provides robust features such as async functions, middleware, and routing. Koa has a minimalistic design, providing developers with a fast and straightforward framework for building web applications.
Koa is built on top of the popular Node.js platform and utilises a modern JavaScript syntax for development. The framework supports asynchronous functions and middleware, allowing developers to write concise and clean code. Koa also offers routing capabilities and an easy way to set up and customise routes.
Koa also provides powerful error-handling capabilities. With Koa, developers can easily create custom error-handling middleware to handle any errors during the development process. This helps developers create robust and stable applications.
Koa also has a built-in debugging tool, making debugging and troubleshooting much easier. The framework is also constantly updated with new features, keeping developers up-to-date with the latest technologies.
Koa is a great choice for developers seeking a powerful, lightweight web development framework. It provides all the features and capabilities needed for modern web application development and is easy to use and customise. With its fast and intuitive design, Koa is ideal for building high-performance, scalable web applications.

Why not Express? It seems more popular.

Koa and Express are popular web frameworks for Node.js. They both offer powerful features that make it easy to create web applications with Node.js, but they also have some distinct differences.
Koa is a newer framework that was released in 2013. It has a smaller API and uses async/await instead of callbacks in the middleware. It is more modern and has the potential to be more efficient. It also has a smaller learning curve and fewer prerequisites.
Express is a more established framework and has been around since 2010. It offers a larger API and a more robust set of features for web development. Express has a larger community, better documentation, and more third-party libraries. It is also easier to learn for beginners, as it has more tutorials available.
In conclusion, both Koa and Express are great options for web development with Node.js. Koa is more modern and efficient, while Express is more established and has a larger community. Depending on the project, one framework may be better suited than the other.

Sample Koa server

Create a new Typescript project, install Koa and related packages as below.

npm i koa koa-bodyparser koa-json koa-logger koa-router @types/koa @types/koa-bodyparser @types/koa-json @types/koa-logger @types/koa-router

Let's create the index.ts with the following contents.

```
import Koa from "koa";
import Router, {RouterContext} from "koa-router";
import logger from "koa-logger";
import json from "koa-json";

const app: Koa = new Koa();
const router: Router = new Router();


router.get('/', async (ctx: RouterContext, next: any) => {
  ctx.body = { msg: 'Hello world!' };
  await next();
})

app.use(json());
app.use(logger());
app.use(router.routes()).use(router.allowedMethods());

app.listen(10888, () => {
  console.log("Koa Started");
})
```

The code is very simple. Here the program importing *koa* and the middlewares that will use. Then we're creating a new Koa app instance and a router instance. We wrote a view handler for the path — */*. Koa now supports **async/await syntax** so we could use an async handler here. We used the fat arrow syntax in this example. The function takes in a koa context and the next middleware in line. We can set the response using the *ctx* object and await the next middleware to respond.
Don't worry if the next() or middleware part doesn't make sense right now. We will discuss middleware in depth later. For the time being, just remember it's important to do this.
After that, the program applied the *json* and *logger* middlewares.

- The json middleware helps in rendering json responses.
- The logger middleware prints useful information on each request.

app.listen starts the app. The program can pass a callback which is executed when the app starts.

Compile your code in a specified folder

From now on, you will have many source codes in a single project. If you just compile the code with *tsc*, there will have mixed up with the source code (TS files) and (JS files) inside your project code.

Therefore, you are required to updated the tsconfig.json before execute the program by uncommenting the line below:

```
"outDir": "./",
```

Please remind to change the path from "./" to a output folder name (such as "./out/")

Since you are going to compile ALL Typescript files inside the project folder, you can just run the command with tsc (no source code specify). All the compiled code will save in the output directory.

Use Postman to access the API once you have executed the output project. The API address is http://localhost:10888

You can press Ctrl-C to shut down the API server.

Parsing Request Body

Let's try some POST / PUT / PATCH requests and learn how to handle the request body, more precisely how we can access the data sent from the client.
To parse request body, we're going to use the *koa-bodyparser* package which can handle several content types. We use the body parser middleware for parsing the incoming payload according to it's content type and store the content in JS data structure on *ctx.request.body*. Let's update the *index.ts* with the middleware and test things out.

```
import Koa from "koa";
import Router, {RouterContext} from "koa-router";
import logger from "koa-logger";
import json from "koa-json";
import bodyParser from "koa-bodyparser";

const app: Koa = new Koa();
const router: Router = new Router();


router.get('/', async (ctx: RouterContext, next: any) => {
  ctx.body = { msg: 'Hello world!' };
  await next();
})

router.post('/', async (ctx: RouterContext, next: any) => {
  const data = ctx.request.body;
  ctx.body = data;
```

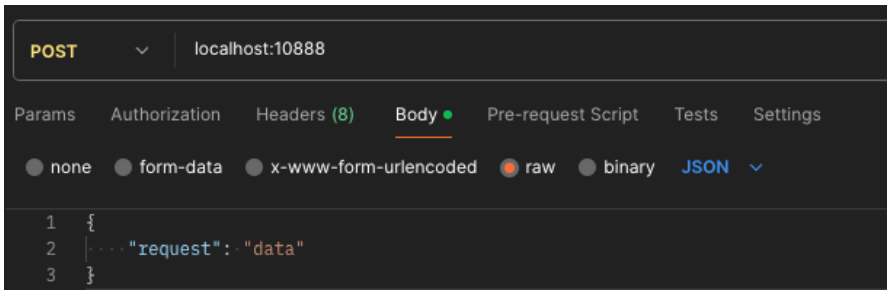
```
  await next();
})

app.use(json());
app.use(logger());
app.use(bodyParser());
app.use(router.routes()).use(router.allowedMethods());

app.listen(10888, () => {
  console.log("Koa Started");
})
```

Now, compile the code again. Test the API through Postman:

1. Set the GET request with http://localhost:10888 and call;
2. Set the POST request with http://localhost:10888. In the Body tab, change the option to "raw" and select "JSON". In the body area, you can input a valid JSON content and try to call it.



Try your skill

Try to create new endpoints /film as below:

1. GET – Return a list of films (more than one)
2. POST – Insert a new film
3. PUT – Update a film

How to implement a GET – Return a film name with a parameter name id? How to get the parameter from the endpoint?
You can define the required parameters by your own.

Validation of the request

Validating requests for an API is an important security measure that helps protect your system from malicious actors. It ensures that data is being sent in the correct format and that all necessary parameters are present. Validation also helps to identify malicious requests, such as those that contain malicious code, or those that attempt to submit sensitive data or access restricted areas. Validating requests helps protect your system from hackers and other malicious actors and can help reduce the risk of data loss or data breaches.

You need to install the addition package koa-req-validation to the project. (*This is a TypeScript framework and no need to install with @types/*). In the source code, you are required to import related modules from koa-req-validation.

```
import { CustomErrorMessageFunction, query, body, validationResults } from
"koa-req-validation";
```

Module Name	Description
CustomErrorMessageFunction	Define the error message if the validation failed
query	Get the values from query string
body	Get the values from body
param	Get the values from endpoint parameter
validationResults	Get the result of the Validation

Define the error message

Next, it is better to predefine some *error* messages. As an example, here is the definition of the Error message callback

```
const customErrorMessage: CustomErrorMessageFunction = (
  _ctx: RouterContext,
  value: string
) => {
  return (
    `The name must be between 3 and 20 ` +
    `characters long but received length ${value.length}`
  );
};
```

Create the validator for each endpoint

Within the definition of each endpoint, you are required to define the validation option:

Original endpoint definition:

```
router.get('/',  async (ctx: RouterContext, next: any) => {
…
  });
```

Updated endpoint definition with validator.

```
router.get('/', [Validators],async (ctx: RouterContext, next: any) => {
…
  });
```

Example:

```
router.get('/',
  query("name")
    .isLength({ min: 3 }).optional()
    .withMessage(customErrorMessage)
    .build(),
  async (ctx: RouterContext, next: any) => {
    …
  });
```

Alternatively, you can also define the validator with an array if there is more than one item to be validated:

```
const validatorName = [
  body("name").isLength({ min: 3
}).withMessage(customErrorMessage).build(),
  body("id").isInt({ min: 10000, max: 20000 }).build()
]
…
router.post('/', ...validatorName, async (ctx: RouterContext, next: any) =>
{
  ...
});
```

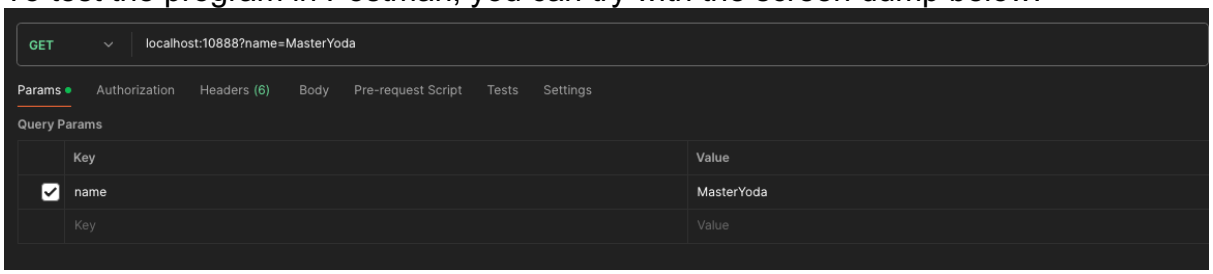
You can check the function names available for the validator definition through the following links:
https://ppeerttu.github.io/koa-req-validation/modules.html#SanitationFunctionName
https://ppeerttu.github.io/koa-req-validation/modules.html#ValidatorFunctionName
At the end of each validator, you need to end with build().

Check with validator and return response with errors

It is simply using if…else… to response the error (422, Unprocessable entity or 400) if the required parameters are not accepted. Example:

```
  const result = validationResults(ctx);
   if (result.hasErrors()) {
     ctx.status = 422;
     ctx.body = { err: result.mapped() }
   } else {
     ctx.body = { msg: `Hello world! ${ctx.query.name}` };
   }
   await next();
```

To test the program in Postman, you can try with the screen dump below:



Try your skill

Update the endpoints you have defined previously with validators.

Response 404 status code if the endpoint does not exist

During to security issue, you should define a response with 404 (Not Found) status code in the API project. This definition MUST declare at the end of all routing endpoints.

```
app.use(async (ctx: RouterContext, next: any) => {
  try {
    await next();
    if (ctx.status === 404) {
      ctx.status = 404;
      ctx.body = { err: "No such endpoint existed" };
    }
  } catch (err: any) {
    ctx.body = { err: err };
  }
})
```

Compile again and try with some endpoints that NOT existed in your source code:

Checkpoint – Git Repository

You are required to create a new file named ".gitignore" which make sure that the listed files/folders are not upload to the GitHub.

You should create a Git repository here (for your portfolio assignment) and upload the laboratory to the Git repository as the version 1 of your project.