

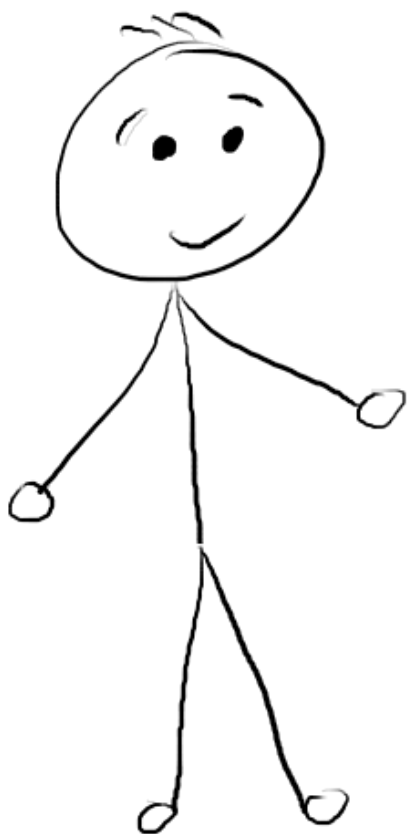


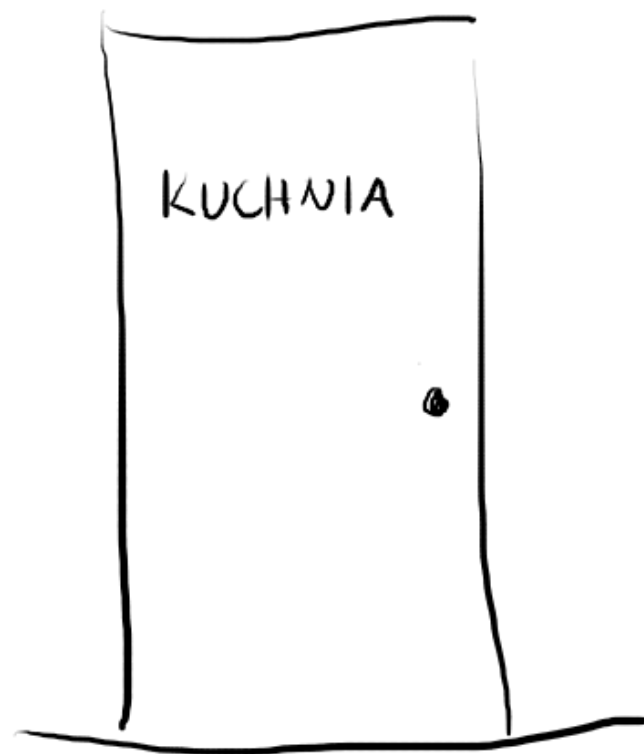
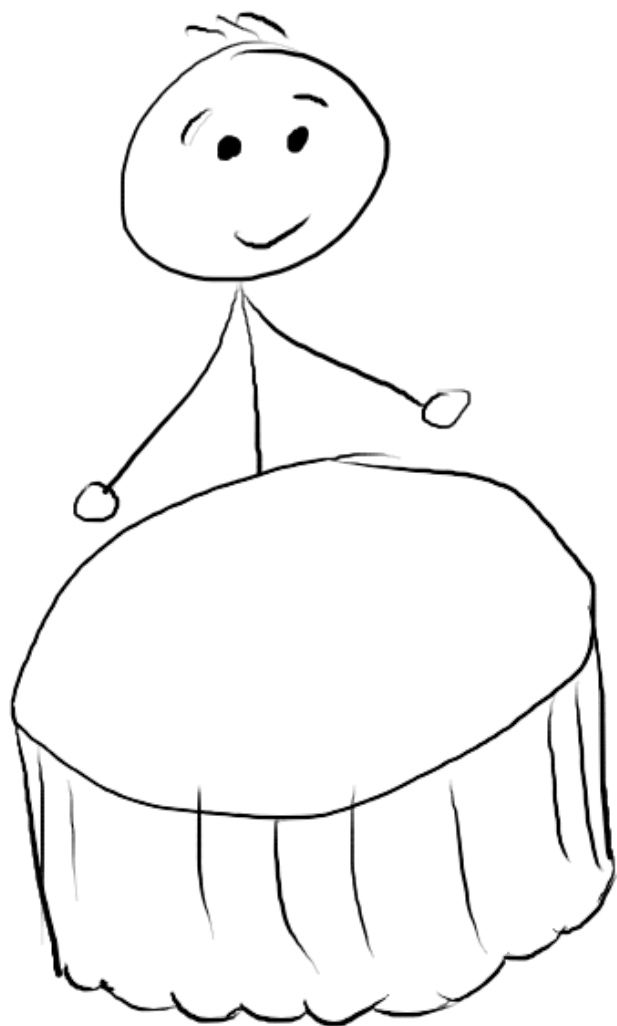
APPLICATION PROGRAMMING INTERFACE

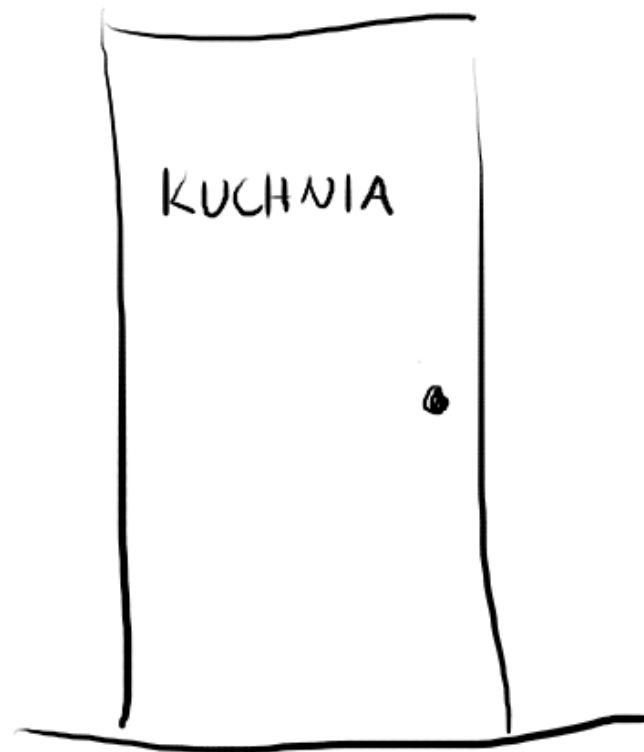
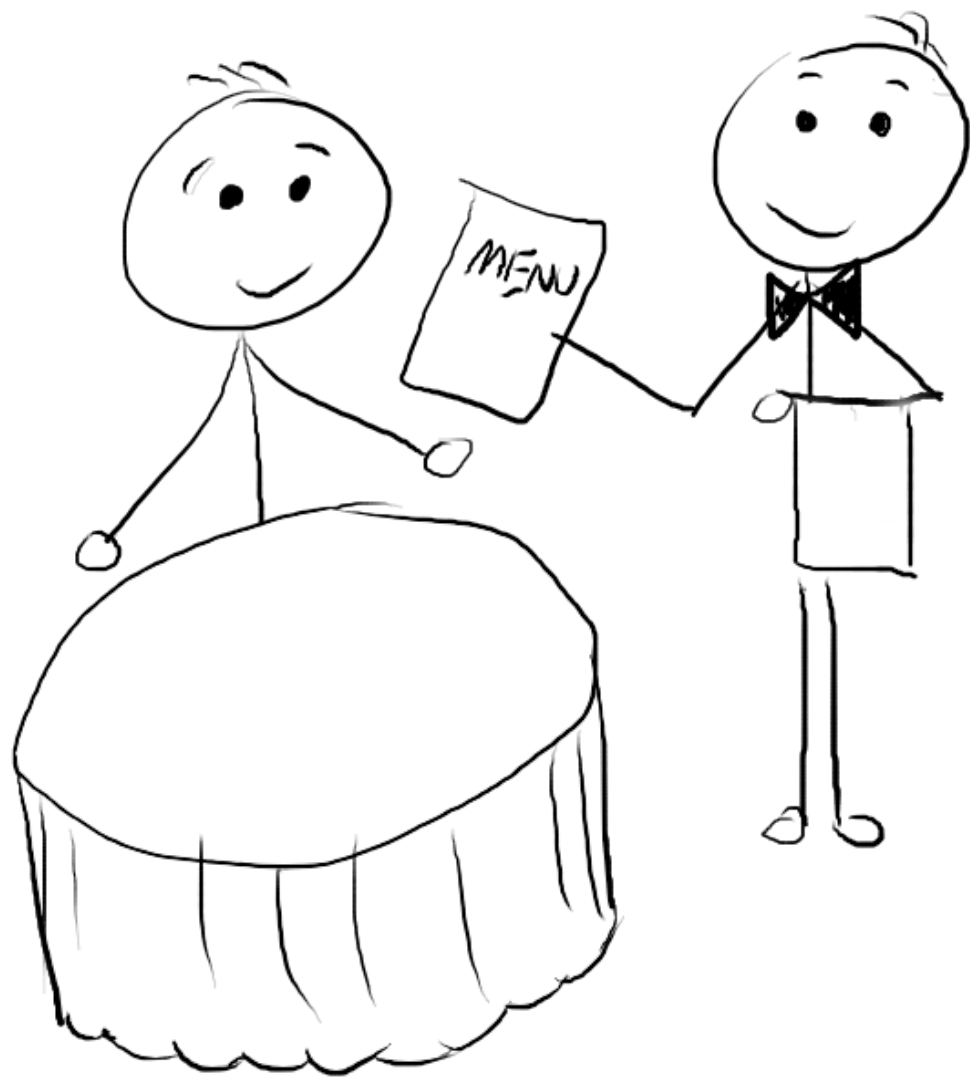
API

Zestaw reguł i ich opis, w jaki programy komputerowe komunikują się między sobą.

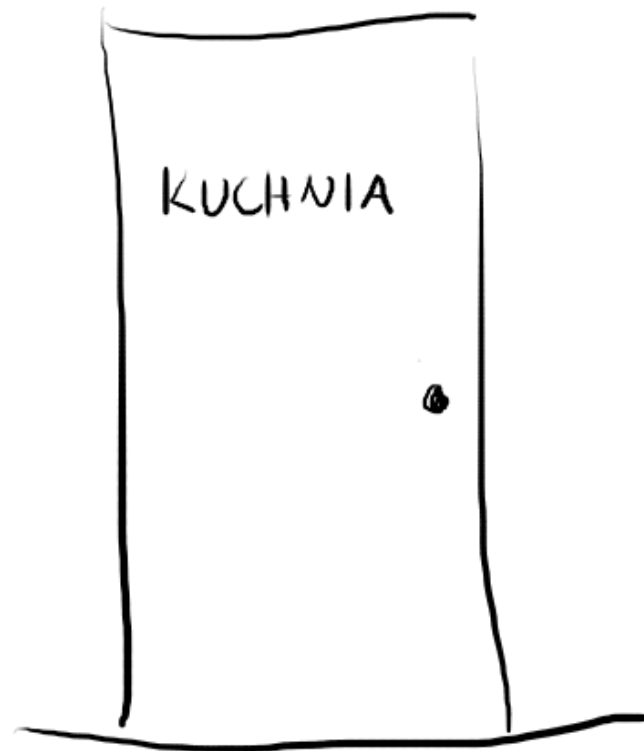
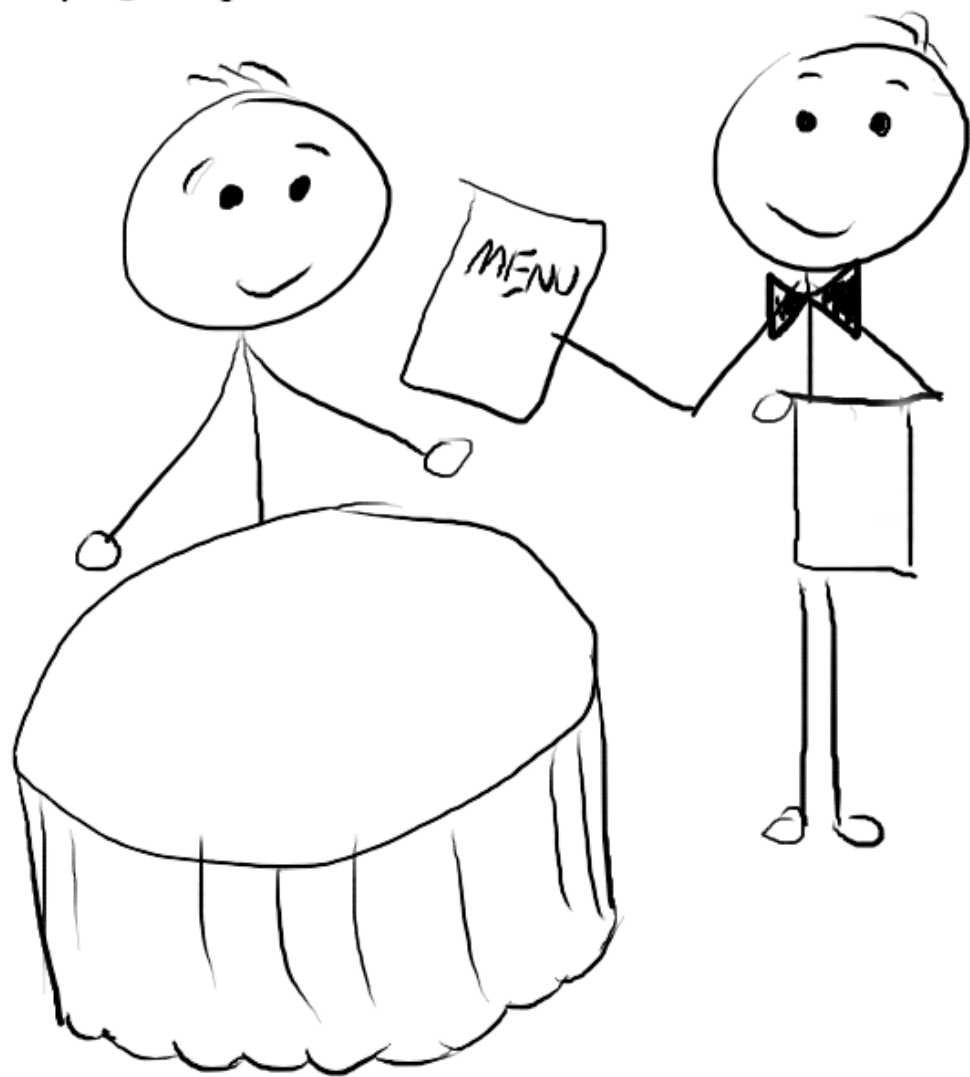
Zadanie interfejsu programowania aplikacji jest dostarczenie odpowiednich specyfikacji podprogramów, struktur danych, klas obiektów i wymaganych protokołów komunikacji.





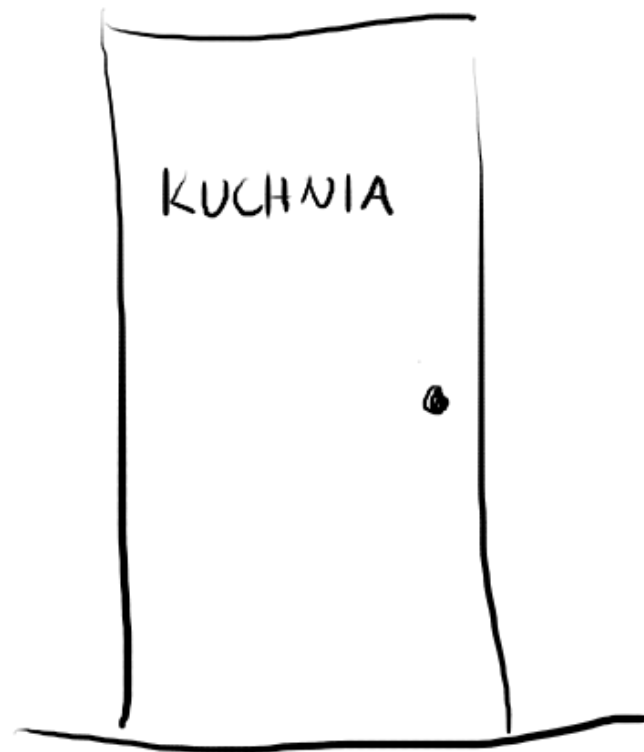
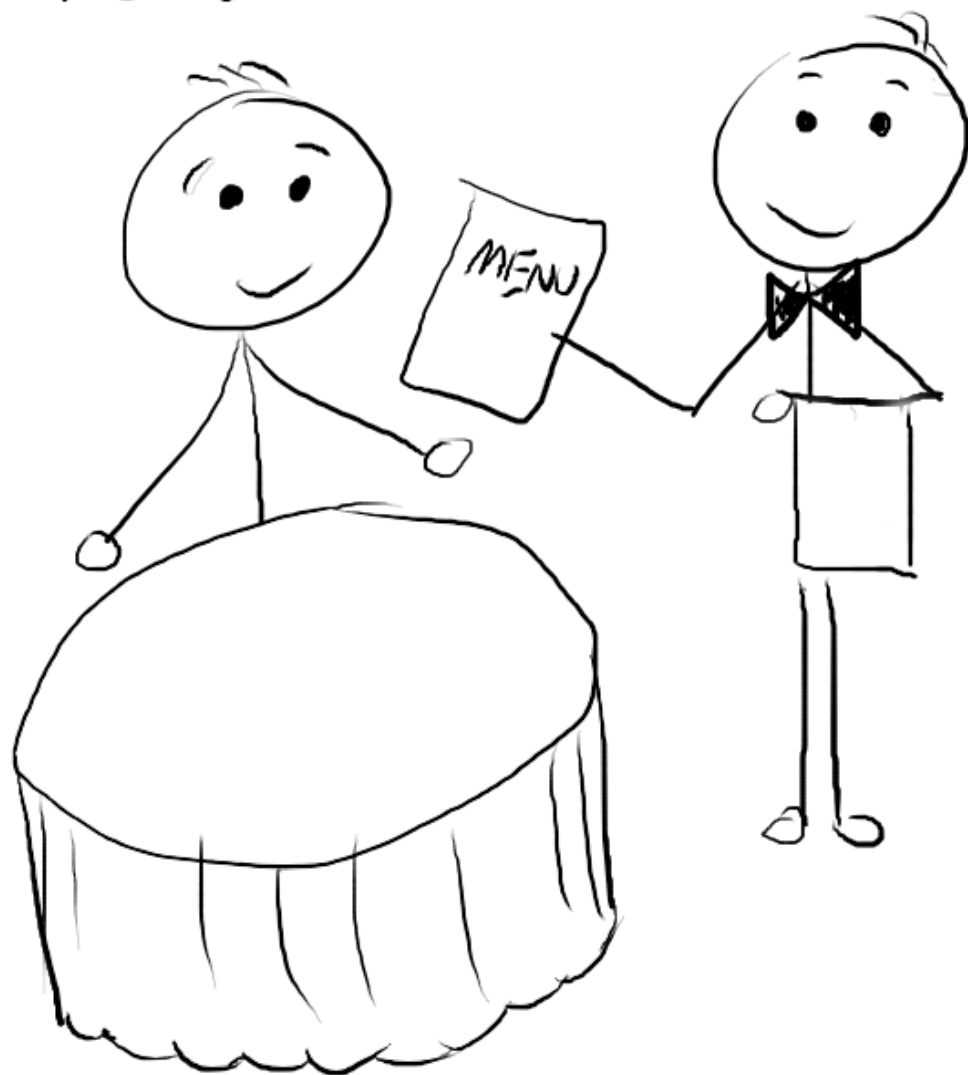


KLIENT

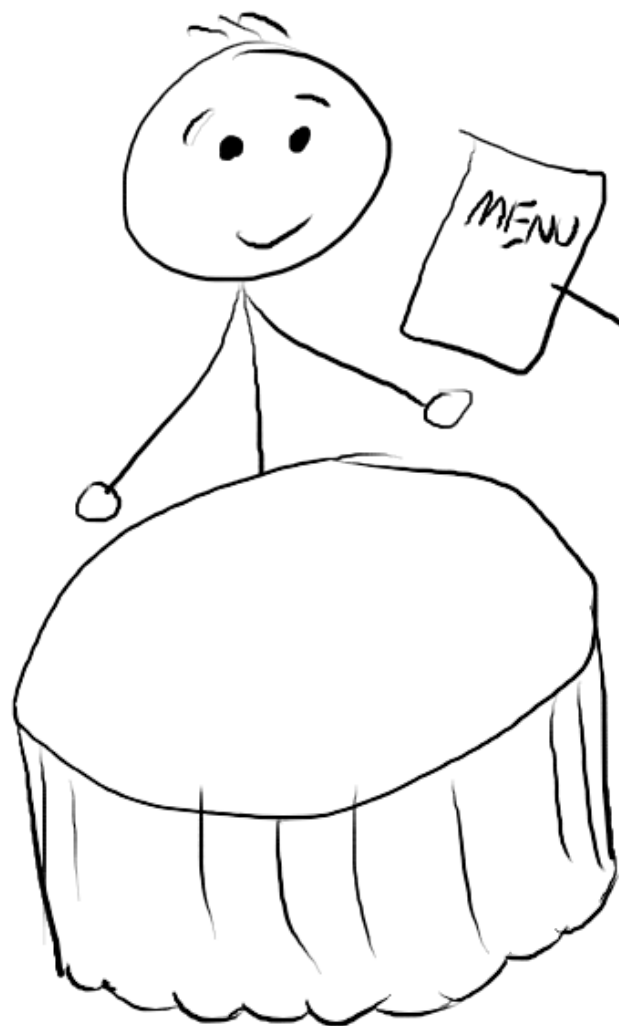


KLIENT

API



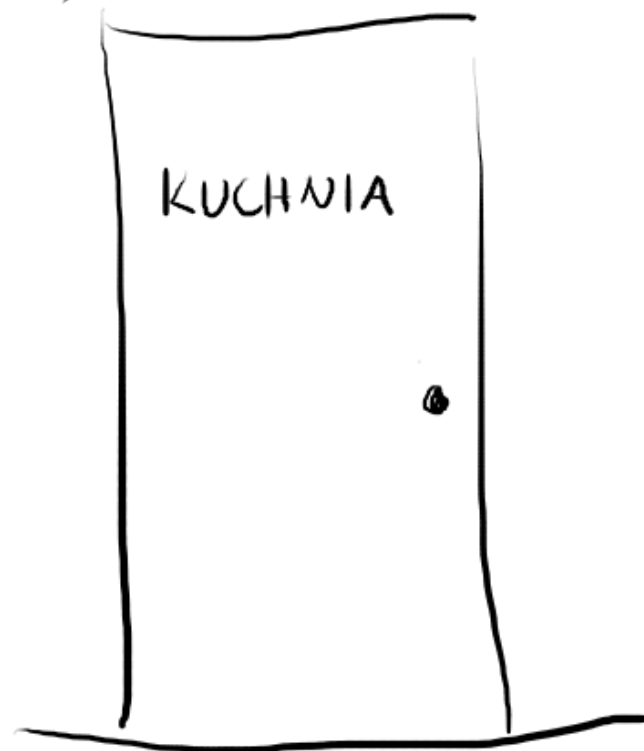
KLIENT

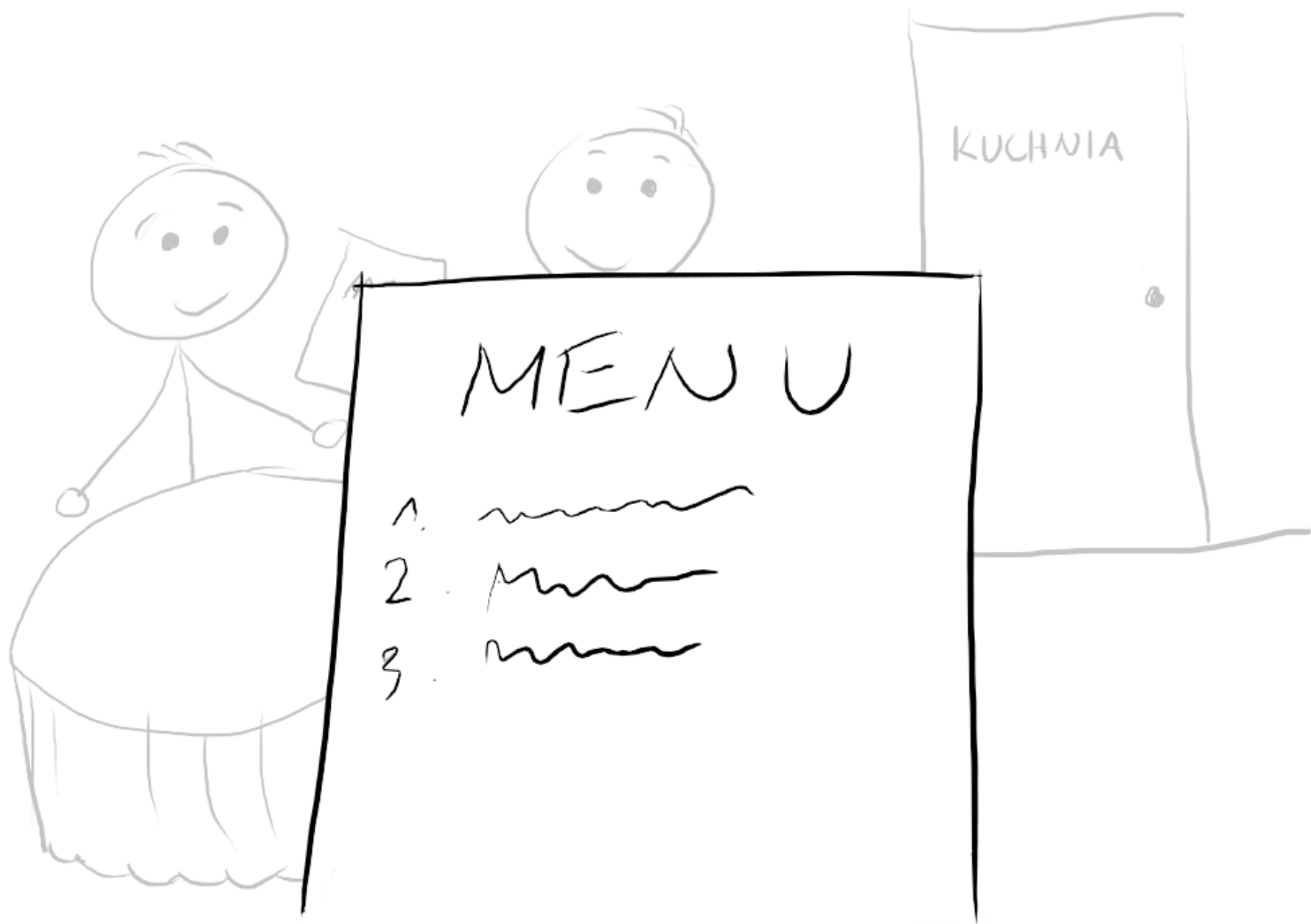


API




SERWER

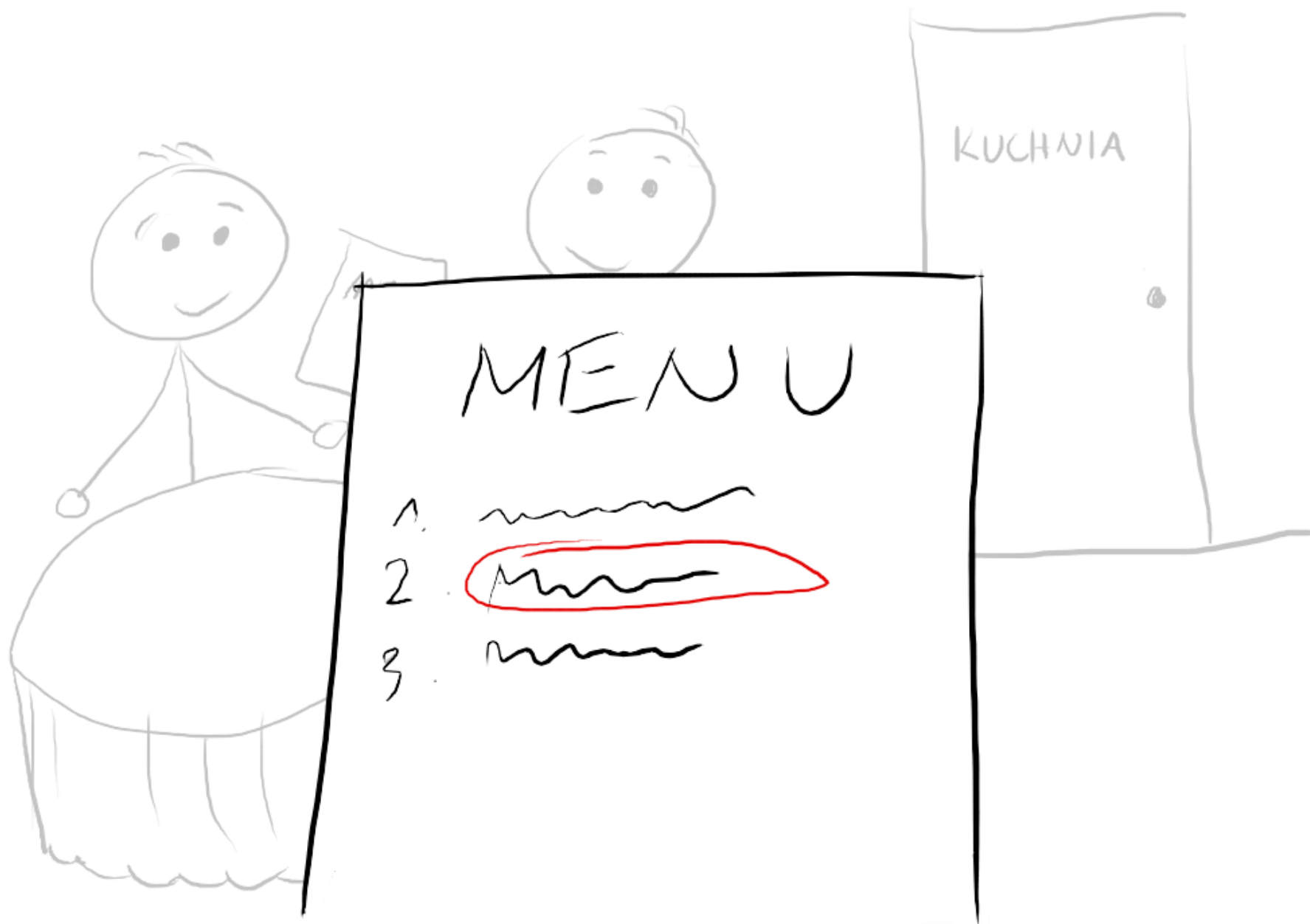




MENU

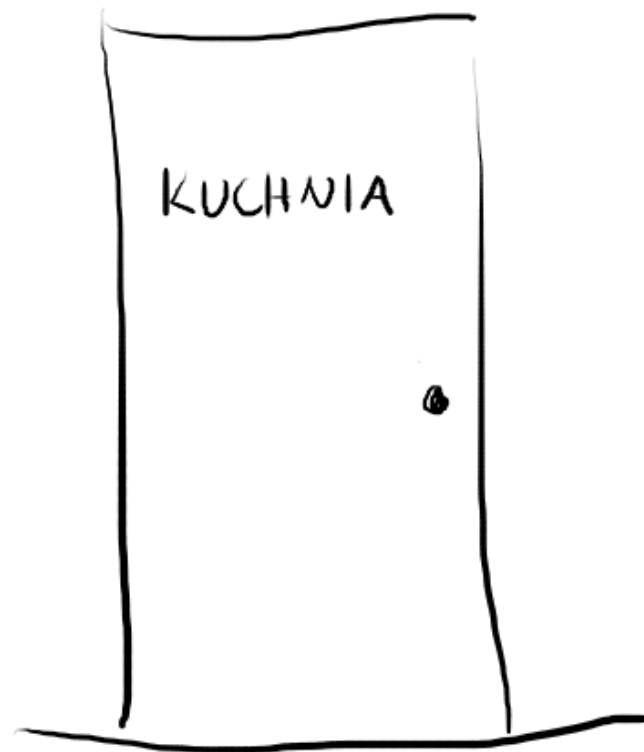
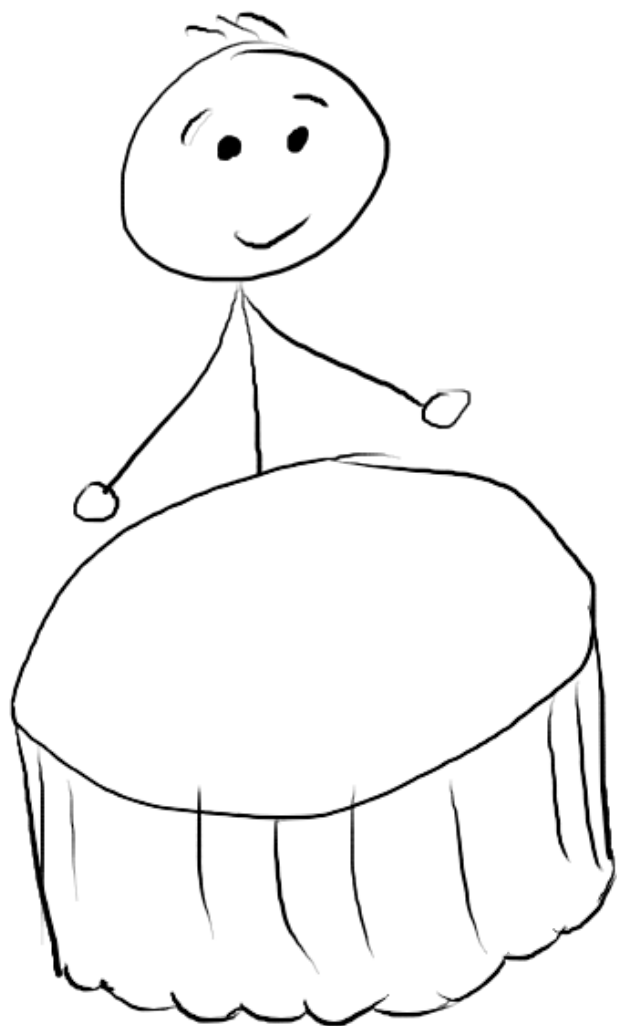
1. 
2. 
3. 

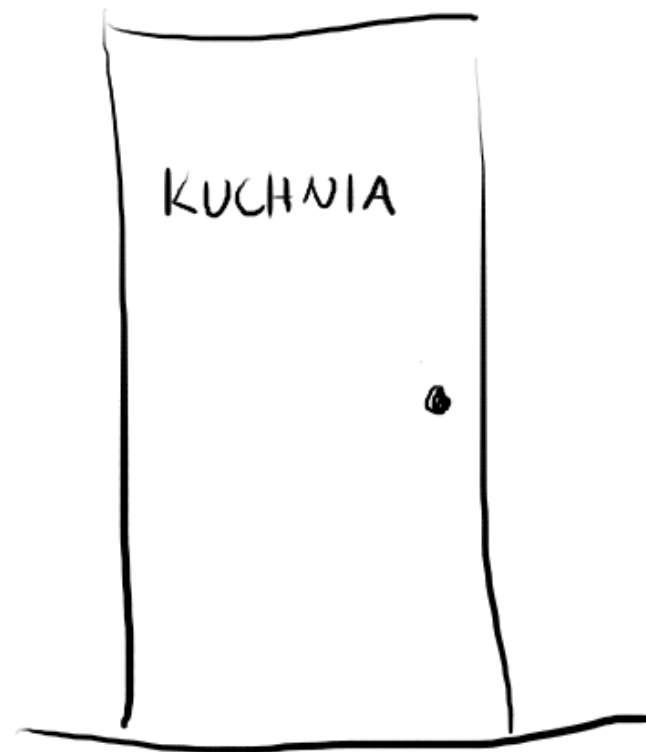
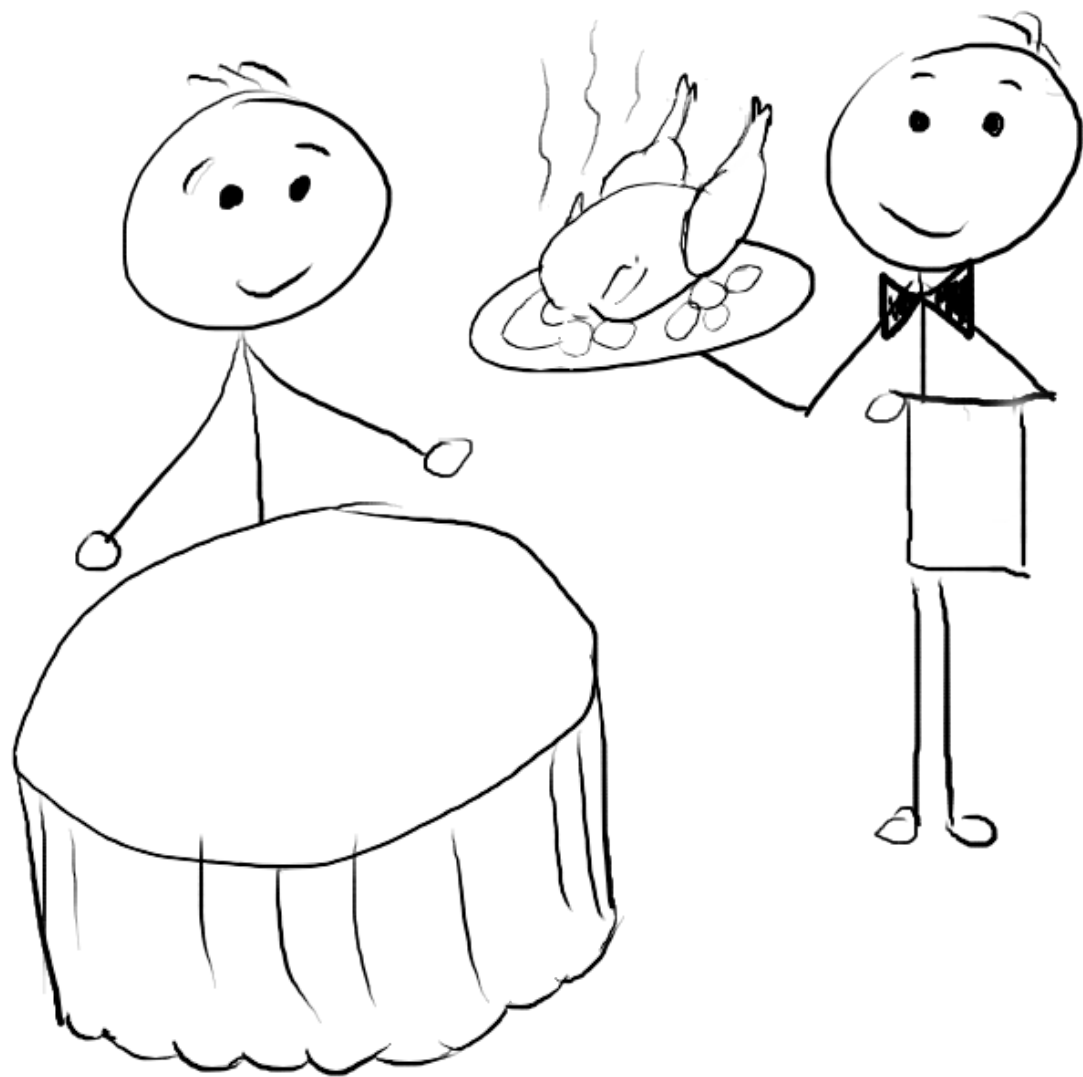
KUCHNIA

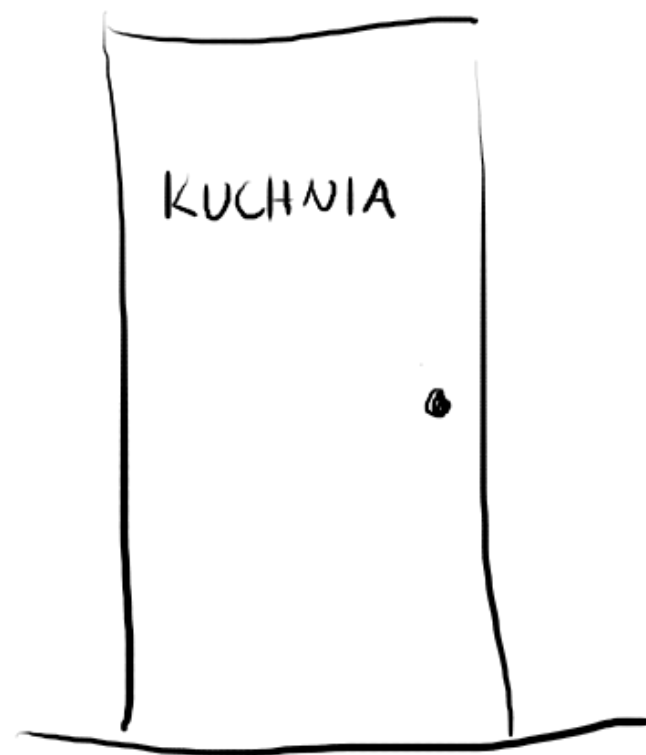
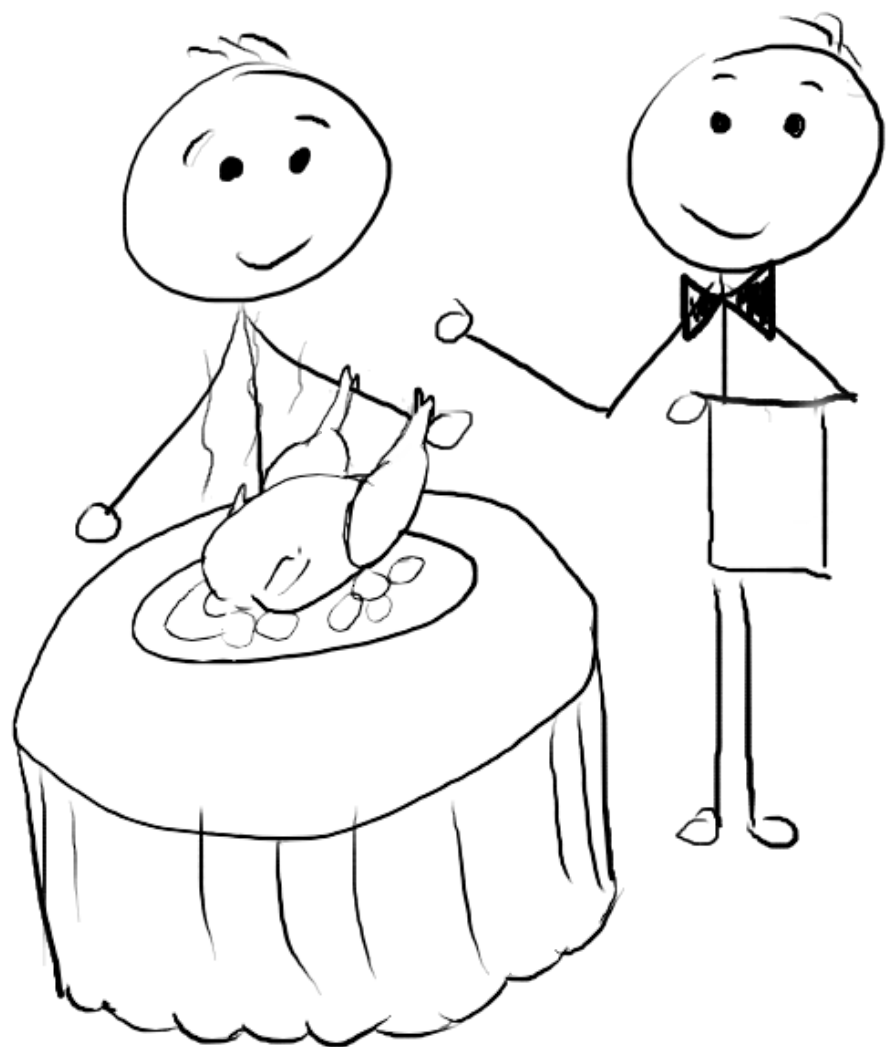


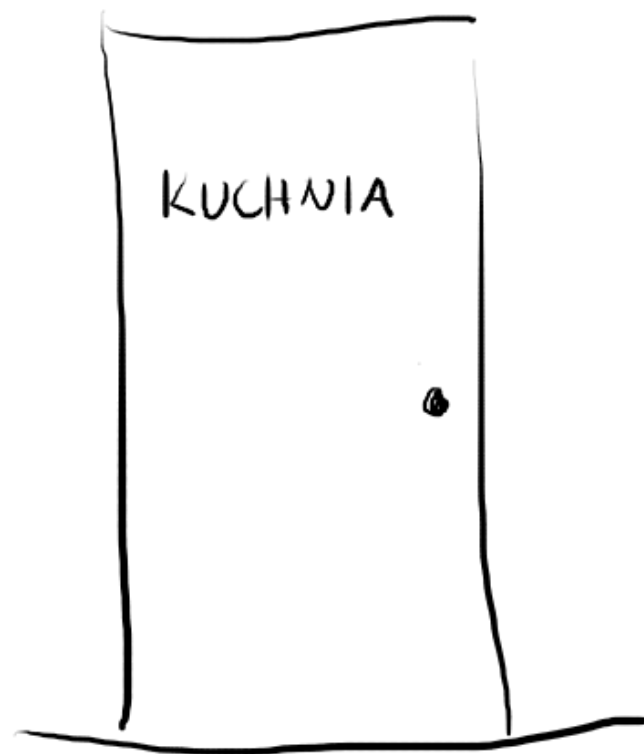
MENU

1. ~~~~~
2. ~~~~~
3. ~~~~~









REMOTE PROCEDURE CALL

RPC

RPC - czyli zdalne wywołanie procedur.

Podejście **RPC** ma wiele znaczeń oraz wiele form. W przypadku web, wywołanie **RPC** pozwala na manipulowanie danymi poprzez protokół **HTTP**.

RPC w rozumieniu web serwera: WYGOPIAO(What You GET Or POST Is An Operation).

ZAPAMIĘTAĆ!

- Struktura komunikacji RPC nie jest z góry ustalona.
- Końcówka powinna zawierać nazwę procedury którą chcemy wywołać na zdalnym serwerze.
- Standard ten przyjmuje tylko 2 metody komunikacji GET oraz POST.

Przykład



```
1 GET /getUsers?someType=abc
```



```
1 POST /saveNewUser
2 {
3     "name": "Jan",
4     "lastName": "Nowak"
5 }
```

STANDARDY BAZUJĄCE NA RPC

- JSON-RPC
- JSON-XML
- SOAP

REPRESENTATIONAL STATE TRANSFER

REST

REST - zmiana stanu poprzez reprezentacje

Standard określający zasady projektowania **API**. Bazuje na protokole **HTTP**.

Za pomocą interfejsu **REST API** możemy eksponować dane jako zasoby, którymi manipulujemy za pomocą odpowiednich metod protokołu **HTTP**.

REST ...

Metody jakie udostępnia nam protokół **HTTP** w łatwy sposób możemy przyporządkować do operacji **CRUD**(*Create/Read/Update/Delete*) na obiekcie.



```
1 C => Create => POST
2 R => Read => GET
3 U => Update => PUT/PATCH
4 D => Delete => DELETE
```

Przykład



```
1 GET /users
2 GET /users/:id
3 POST /users
4 PUT /users/:id
5 DELETE /users/:id
```


OGÓLNE ZASADY REST

- **Uniform interface** - Interfejs powinien zapewnić ustandaryzowaną komunikację pomiędzy klientem a serwerem
- **Client-server** - Wyraźny podział na aplikację po stronie klienta i serwera
- **Stateless** - czyli każde zapytanie powinno zawierać komplet informacji do poprawnego obsłużenia żądania

OGÓLNE ZASADY REST ...

- **Cacheable** - API powinno wspierać cache'owanie danych w celu zwiększenia wydajności
- **Layered system** - klient łączący się do serwera nie powinien wiedzieć co się dzieje po drugiej stronie
- **Code on demand(*)** – API przewiduje możliwość wysłania fragmentu kodu, który może być wykonany po stronie klienta

RESOURCE

Resource – czyli zasób. Dowolna informacja, która posiada nazwę może być zasobem jeżeli:

- jest rzeczownikiem, np.: user, post, comment
- jest unikatowa i wskazuje na konkretną rzecz
- może być przedstawiona w formie danych
- posiada przynajmniej jeden adres URI, pod którym jest dostępny

NAZEWNICTWO

Zasoby powinny być tworzone w taki sposób, aby reprezentowały obiekt. Dzięki temu możliwe jest wykonanie wielu akcji na pojedynczym zasobie.



```
1 GET /users
2 POST /users
3 PATCH /users
4 DELETE /users/:id
```



```
1 GET /getUsers
2 POST /addUsers
3 POST /deleteUser?id=:id
```

REPREZENTACJA

- JSON
- YAML
- XML
- ...

GRAPHQL

GRAPHQL

GraphQL jest językiem zapytań dla interfejsów API i środowiskiem wykonawczym do wypełniania zapytań z istniejącymi danymi.

Zapewnia kompletny i zrozumiały opis danych w API.

Pozwala na pobranie wielu zasobów w jednym zapytaniu.

GRAPHQL

GraphQL daje możliwość zadawanie zapytań o to czego konkretnie dany klient potrzebuje, bez dodatkowych danych.

Łączy się do jednego adresu do którego wysyła odpowiednie zapytanie, które jest przetwarzane przez serwer.

GRAPHQL

HTTP SERVER

WBUDOWANY MODUŁ HTTP

```
● ● ●  
  
1 const http = require('http');  
2  
3 const server = http.createServer((req, res) => {  
4   res.writeHead(200, {  
5     'Content-Type': 'text/plain'  
6   });  
7   res.end('...');  
8 });  
9  
10 server.listen(4000);
```

WEB FRAMEWORKS

- [express](#) (47 219)
- [koa](#) (28 419)
- [nestjs](#) (24 024)
- [fastify](#) (13 407)
- [connect](#) (8 641)
- ...

Express

Express

Express jest to web framework, który zapewnia solidny zestaw funkcji dla aplikacji internetowych i mobilnych przy minimalnej i elastycznej strukturze.

Hello world!



```
1 const express = require('express');
2 const app = express();
3
4 app.get('/', (req, res) => {
5   res.send('hello world!');
6 });
7
8 app.listen(4500, () => console.log('server started'));
```

BASIC ROUTING

Routing to określanie, w jaki sposób aplikacja odpowiada na żądanie klienta do określonego punktu końcowego i konkretnej metodzie żądania HTTP (GET, POST itd.).

Każda trasa może mieć jedną lub więcej funkcji obsługi, które są wykonywane po dopasowaniu ścieżki.

ROUTING[STRUKTURA]

```
1 app.METHOD(PATH, HANDLER)
```

app - instancja naszego serwera

METHOD - metoda żądania HTTP (małymi literami)

PATH - ścieżka na serwerze

HANDLER - funkcja wykonywana po dopasowaniu ścieżki

ROUTE METHODS



```
1 app.get('/', (req, res) => {  
2   res.send('hello world!');  
3 });  
4  
5 app.post('/', (req, res) => {  
6   res.send('Got a POST request');  
7 });  
8  
9 app.all('/', (req, res) => {  
10  res.send('Any of HTTP method');  
11 });
```

ROUTE PATHS

Ścieżki w połączeniu z metodą żądania, definiują punkty końcowe.

Ścieżki mogą przyjmować zwykły adres(string), wzór(string patterns) lub wyrażenie regularne(RegExp).

Znaki ?, +, * i () są podzbiorami ich odpowiedników w wyrażeniach regularnych. Łącznik (-) i kropka (.) Interpretowane są dosłownie według ścieżek opartych na łańcuchach.

PRZYKŁAD ZWYKŁEJ ŚCIEŻKI

```
● ● ●  
  
1 app.get('/users', (req, res) => {  
2   // ...  
3 });  
4  
5 app.post('/posts.txt', (req, res) => {  
6   // ...  
7 });  
8  
9 app.delete('/comments.json', (req, res) => {  
10  // ...  
11 });
```

ŚCIEŻKI BAZUJĄCE NA WZORZE

```
1 // matches: user, users
2 app.get('/users?', (req, res) => {
3   // ...
4 });
5
6 // matches: users, userss, usersss, ...
7 app.post('/users+', (req, res) => {
8   // ...
9 });
10
11 // matches: users, usxxxers, usRANDOMers
12 app.delete('/us*ers', (req, res) => {
13   // ...
14 });
```

WYRAŻENIE REGULARNE



```
1 // matches: file.txt, abc/kot.txt
2 app.get(/.*\.txt/, (req, res) => {
3   // ...
4 });
5
6 // matches: ala, alaMaKota ...
7 app.post(/^ala.*/ (req, res) => {
8   // ...
9 });
```

ROUTE PARAMETERS

Parametry trasy to nazwane segmenty adresów URL, które służą do przechwytywania wartości określonych na ich pozycji w adresie URL.

Przechwycone wartości są zapełniane w obiekcie `req.params`, a nazwa parametru trasy jest określona w ścieżce jako odpowiadające im klucze.

Przykład



```
1 // Path: /users/:userId/posts/:postId
2 // URL: http://localhost:4500/users/12/posts/44
3
4 app.get('/users/:userId/posts/:postId', (req, res) => {
5   // req.params: { "userId": "12", "postId": "44" }
6 });
```


Przykład



```
1 Path: /getFile/:filename.:extension
2 URL: http://localhost:4500/getFile/somefile.txt
3
4 req.params: { "filename": "somefile", "extension": "txt" }
```

ROUTE HANDLERS

Route może posiadać wiele funkcji zwrotnych, które wykonują się sekwencyjnie aż zostanie wywołanie wysłania odpowiedzi do klienta. Warunkiem jest to iż funkcje pośrednie zamiast kończyć odpowiedź użyją funkcji callback next.

Procedury obsługi tras mogą mieć postać funkcji, szeregu funkcji lub kombinacji obu.

SINGLE CALLBACK



```
1 app.get('/ala-ma-kota', (req, res) => {  
2   // ...  
3 });
```

MANY CALLBACKS

```
1 app.get(  
2   '/ala-ma-kota',  
3   (req, res, next) => { ...; next() },  
4   (req, res, next) => { ...; next() },  
5   (req, res) => { ... }  
6 );
```

COMBINE CALLBACKS



```
1  const callback1 = (req, res, next) => { ...; next() }
2  const callback2 = (req, res, next) => { ...; next() }
3
4  app.get(
5    '/ala-ma-kota',
6    [ callback1, callback2 ],
7    (req, res) => { ... }
8  );
```

Response methods

- `res.download()` - wyślij plik do pobrania
- `res.end()` - zakończ żądanie
- `res.json()` - wyślij odpowiedź JSON
- `res.redirect()` - przekieruj żądanie

Response methods...

- `res.render()` - wyrenderuj widok
- `res.send()` - wyślij odpowiedź różnych typów
- `res.sendFile()` - wyślij plik w postaci strumienia
- `res.sendStatus()` - ustaw i wyślij kod odpowiedzi jako treść

APP.ROUTE()

```
1 app.route('/users')
2   .get((req, res) => {
3     // ...
4   })
5   .post((req, res) => {
6     // ...
7   })
8   .delete((req, res) => {
9     // ...
10  });
```


EXPRESS.ROUTER

Klasa *express.Router* służy do tworzenia modułowych, zbiorów procedur obsługi ścieżek.

Instancja routera to kompletne oprogramowanie wraz z systemem routingu. Określana również jako mini aplikacja.

Przykład

```
1 // ./dashboard.js
2 const express = require('express');
3 const router = express.Router();
4
5 router.use((req, res, next) => {
6   console.log('time: ', Date.now());
7   next();
8 });
9 router.get('/', (req, res) => {
10   res.send('hello world!');
11 });
12
13 module.exports = router;
```

```
1 // ./app.js
2 const express = require('express');
3 const dashboard = require('./dashboard');
4 const app = express();
5
6 app.use('/dashboard', dashboard);
7
8 app.listen(4500, () => console.log('server started'));
```