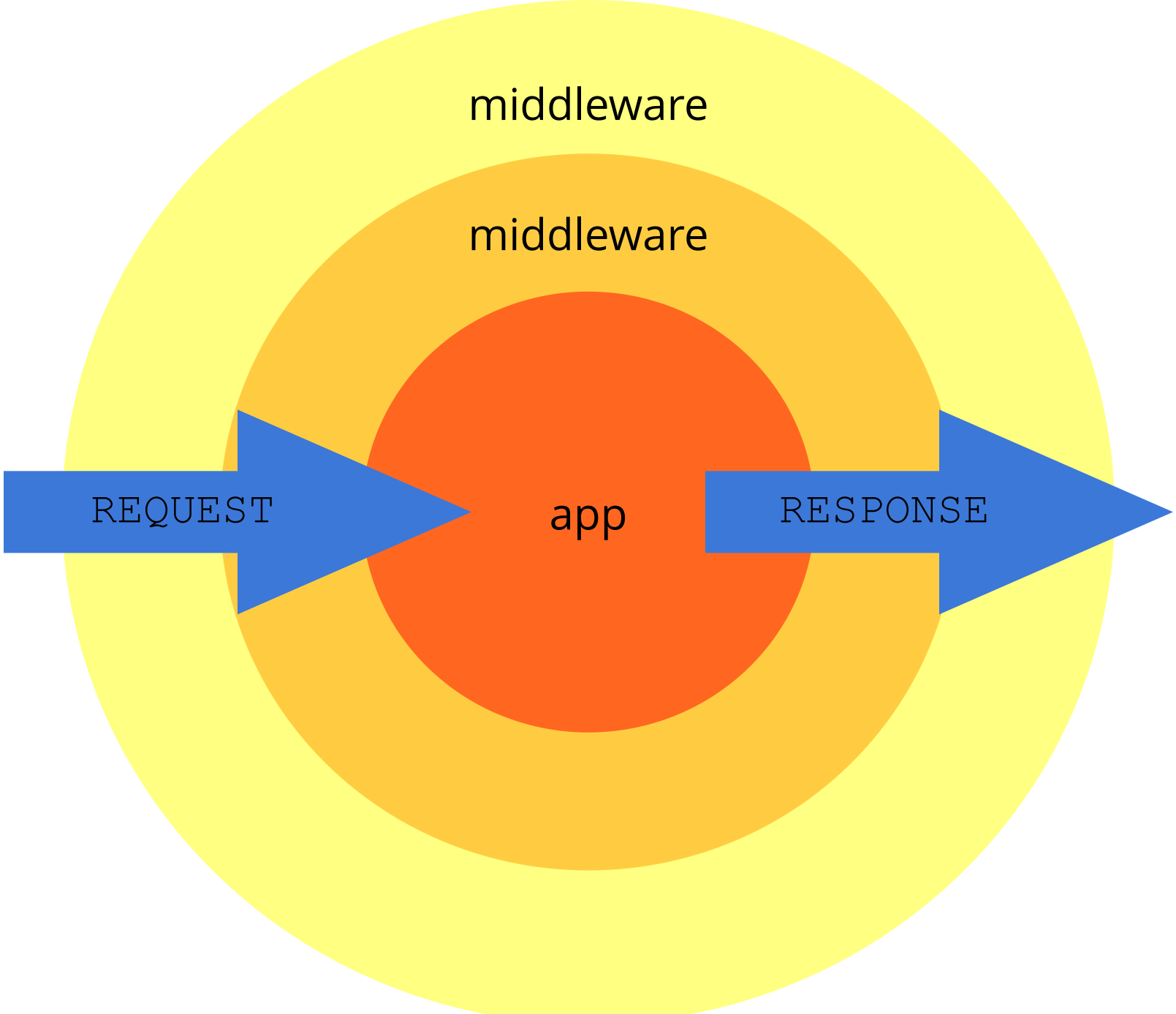

node
JS

Middleware

Middleware

MIDDLEWARE czyli oprogramowanie pośredniczące jest to rodzaj oprogramowania/funkcji umożliwiającej komunikację pomiędzy różnymi aplikacjami, usługami lub systemami.


middleware
middleware
REQUEST
app
RESPONSE

ExpressJS - Middleware

Middleware w Express pełni rolę pośrednią przed wykonaniem odpowiedniej instrukcji dla danej ścieżki.

W funkcji mamy dostęp do żądania i odpowiedzi HTTP oraz funkcji callback next, która odpowiada za wywołanie kolejnego middleware lub przejścia do odpowiedniej instrukcji dla danej ścieżki.

ExpressJS - Middleware

Funkcje middleware mogą wykonywać następujące zadania:

- wykonać dowolny kod
- wprowadzić zmiany w obiektach żądań i odpowiedzi
- zakończyć żadanie
- wywołac następne oprogramowanie pośrednie w stosie

Konstrukcja middleware

```
1  const express = require('express');
2  const app = express();
3
4  const customMiddleware = (req, res, next) => {
5      // some logic ...
6      next();
7  };
8
9  app.use(customMiddleware);
10
11 app.listen(4500, () => console.log('server started'));
```

Zapamiętać!

Jeżeli **middleware** nie kończy żądania/odpowiedzi, musi zostać wywyłana funkcja **next()**, aby przekazać sterowanie do następnej funkcji oprogramowania pośredniego.

W przeciwnym wypadku żądanie zostanie zawieszone.

Przykład

```
// ...
const timeMiddleware = (req, res, next) => {
    req.requestTime = new Date();
    next();
};

app.use(timeMiddleware);

app.get('/', (req, res) => {
    res.send('request time: ' + req.requestTime);
});
// ...
```

Typy middleware

W Express możemy wykorzystać takie typy middleware jak:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

APPLICATION-LEVEL MIDDLEWARE

Powiązanie oprogramowania warstwy pośredniej na poziomie aplikacji.

Ten rodzaj middleware będzie wykonywał się dla całej aplikacji lub dla wydzielonej grupy ścieżek.

Przykład

```
1 const express = require('express');
2 const app = express();
3
4 app.use((req, res, next) => {
5     console.log('current time', new Date());
6     next();
7 });
8
9 app.listen(4500, () => console.log('server started'));
```

Przykład ze ścieżką

```
1 const express = require('express');
2 const app = express();
3
4 app.use('/user', (req, res, next) => {
5     req.userTime = new Date();
6     next();
7 });
8
9 app.listen(4500, () => console.log('server started'));
```

ROUTER-LEVEL MIDDLEWARE

Router-level middleware działa podobnie jak application-level middleware jedyna różnica jest taka iż ten rodzaj middleware działa w obrembie instancji express.Router().

Przykład

```
1 const express = require('express');
2 const router = express.Router()
3
4 router.use((req, res, next) => {
5     console.log('current time', new Date());
6     next();
7 });
8 // ...
```

ERROR-HANDLING MIDDLEWARE

Error-handling middleware obsługuje błędy oprogramowania. Ten rodzaj middleware przyjmuje cztery argumenty.

Definicja tego rodzaju middleware wygląda podobnie jak w przypadku wcześniejszych typów.

Przykład

```
1 const express = require('express');
2 const app = express();
3
4 app.use((error, req, res, next) => {
5     console.error(error.message);
6     res.send(500, error.message);
7 });
8 // ...
```

BUILT-IN MIDDLEWARE

Built-in middleware czyli wbudowane w webframework Express oprogramowanie pośrednie.

- **express.static** - middleware odpowiedzialny za udostępnianie statycznych zasobów
- **express.json** - parsowanie zawartości przychodzącego z żądania typu JSON
- **express.urlencoded** - parsowanie zawartości przychodzącego z żądania z formularza

BUILT-IN MIDDLEWARE

Od wersji 4.x, Express udostępnia jedynie trzy wcześniej wymienione funkcje pośrednie.

Reszta middleware jest rozwijana w odzielnych modułach.

Lista dostępnych modułów rozwijanych przez Express

THIRD-PARTY MIDDLEWARE

Third-party middleware czyli funkcje pośrednie stworzone przez innych poza zespołem twórców framework'a Express.

Przykład

```
1 const express = require('express');
2 const bodyParser = require('body-parser'):
3 const app = express();
4
5 app.use(bodyParser.json());
6
7 // ...
```

Template

Template

Express udostępnia mechanizm, który umożliwia korzystanie ze statycznych plików szablonów w aplikacji.

W środowisku wykonawczym silnik szablonów zastępuje zmienne w pliku statycznym, rzeczywistymi wartościami i przekształca szablon w plik HTML wysyłany do użytkownika.

Template engines

Niektóre popularne silniki szablonów współpracują z Express'em(np. JADE/Pug, Mustache, EJS).

Express domyślnie używa JADE/Pug jako silnik szablonów.

Pełna lista silników szablonów

Render

Aby przerenderować pliki szablonów niezbędna jest odpowiednia konfiguracja:

- ustawić ścieżkę do katalogu gdzie będą znajdować sie szablony, np: **app.set('views', './templates');**
- ustawić odpowiedni silinik szablonów, np: **app.set('view engine', 'pug')** oraz pobrać odpowiednią paczkę z npm

Przykład

```
// npm install pug

const express = require('express');
const app = express();

app.set('view engine', 'pug')
app.get('/', function (req, res) {
    const scope = { title: 'some title', header: 'heloo!' };
    res.render('index', scope);
});
// ...
```

PUG template

```
1 html
2     head
3         title= title
4     body
5         h1= header
```

Przykład (react)

```
// npm install express-react-views react react-dom
const express = require('express');
const app = express();
const reactViews = require('express-react-views')

app.set('view engine', 'jsx');
app.engine('jsx', reactViews.createEngine());

app.get('/', function (req, res) {
    const scope = { name: 'Jan' };
    res.render('hello', scope);
});

app.listen(4500, () => console.log("server started"));
```

views/hello.jsx

```jsx
const React = require('react');

const HelloMessage = (props) => (<div>Hello {props.name}</div>);

module.exports = HelloMessage;
```

Error handling

Error handling

Obsługa błędów odnosi się do sposobu, w jaki framework Express przechwytuje i przetwarza błędy, które występują synchronicznie jaki i asynchronicznie.

Express sam w sobie zawiera domyślną procedurę obsługi błędów, więc jeżeli nie ma takiej potrzeby to nie trzeba jej pisać samodzielnie.

Catching errors

Ważne jest, aby program Express przechwytywał wszystkie błędy występujące podczas uruchamiania programów do obsługi tras jak i middleware.

Błędy występujące w kodzie synchronicznym wewnątrz procedur obsługi tras i oprogramowania pośredniego nie wymagają dodatkowej pracy. Jeśli kod synchroniczny zgłasza błąd, Express przechwytuje go i przetwarza.

Przykład synchroniczny

```
1 const express = require('express');
2 const app = express();
3
4 app.get('/', function (req, res) {
5     throw new Error('some dumb error');
6 });
7 // ...
```

Catching errors

W przypadku błędów zwracanych przez funkcje asynchroniczne wywoływane przez procedury obsługi i oprogramowanie pośrednie, należy przekazać je do funkcji next(), w której to Express je przechwyci i przetworzy.

Przykład asynchroniczny

```
1  const express = require('express');
2  const app = express();
3
4  app.get('/', function (req, res) {
5      setTimeout(() => {
6          const error = new Error('some dumb error');
7          next(error);
8      }, 5000);
9  });
10 // ...
```

Default error handler

Express dostarcza wbudowaną obsługę błędów, która dba o wszelkie błędy, które mogą wystąpić w aplikacji.

Ta domyślna funkcja obsługi oprogramowania do obsługi błędów jest dodawana na KOŃCU stosu funkcji oprogramowania pośredniego.

Default error handler

Jeśli przekażemy błąd do funkcji next() i nie obsłużymy go w niestandardowej procedurze obsługi błędów, zostanie ona obsłużona przez wbudowany mechanizm obsługi błędów.

Error handlers

Konstrukcja middleware do obsługi błędów przyjmuje cztery argumenty, nie tak jak w przypadku zwykłych funkcji pośrednich. (err, req, res, next)

Middleware zawiązane z obsługą błędów muszą być użyte po dodaniu innych funkcji pośredniczących jak i obsługi ścieżek.

Przykład

```
// ...
app.use(bodyParser.json());
app.use(cookieParser);

app.get('/', ...);
/// ...

app.use((error, req, res, next) => {
    // ...
});
// ...
```