



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH***

Szkielety programistyczne w aplikacjach internetowych

dr Mariusz Dzieńkowski

Lublin 2022

LABORATORIUM 9. ZAGROŻENIA I OCHRONA APLIKACJI INTERNETOWYCH

Cel laboratorium:

Identyfikacja głównych zagrożeń dla aplikacji w sieci Internet oraz podstawowe sposoby zabezpieczeń.

Zakres tematyczny zajęć:

- Wykorzystanie protokołu HTTPS oraz szyfrowania połączeń w aplikacjach internetowych.
- Praktyczne dodanie do serwera Express certyfikatu wygenerowanego przez Let's Encrypt.
- Zabezpieczenie przed fałszowaniem żądań przesyłanych między witrynami – atak Cross-Site Request Forgery (CSRF).
- Ochrona przed atakami typu Cross-Site Scripting (XSS).

Pytania kontrolne:

- a) Czym się różni protokół HTTPS od HTTP?
- b) Na czym polega atak sieciowy typu „cookie hijacking”?
- c) Podaj sposób zabezpieczenia aplikacji www przed fałszowaniem żądań przesyłanych między witrynami.

Zadanie 9.1. Implementacja aplikacji wykorzystującej protokół HTTPS oraz szyfrowane połączenia

HTTPS to protokół HTTP z SSL/TLS, który daje uwierzytelnienie dzięki posiadaniu kluczy i certyfikatów, gwarantuje pewnego rodzaju prywatność i poufność (połączenie szyfrowane jest w sposób asymetryczny) oraz integralność danych, ponieważ przesyłane dane nie mogą zostać zmienione podczas tranzytu.

Aplikacja realizowana w ramach tego zadania pokazuje jak praktycznie dodać do serwera Express certyfikat wygenerowany przez Let's Encrypt.

Domyślnie Node.js dostarcza treści przez HTTP, choć istnieje także moduł HTTPS, którego należy użyć, aby komunikować się z klientem przez bezpieczny kanał. Jest to moduł wbudowany, a sposób jego użycia jest bardzo podobny do tego, w jaki korzysta się z modułu HTTP (Listing 9.1).

Listing 9.1 Użycie modułu HTTPS

```
const express = require("express")
const https = require("https")
fs = require("fs")

const options = {
  //key: fs.readFileSync("/srv/www/keys/my-site-key.pem"),
  //cert: fs.readFileSync("/srv/www/keys/chain.pem")
}

const app = express()
```



```
app.use((req, res) => {
  res.writeHead(200)
  res.end("hello world\n")
})

app.listen(8000)
https.createServer(options, app).listen(8080)
```

W kodzie na Listing 9.1 zakomentowana jest część z plikami będącymi certyfikatami SSL, które trzeba wygenerować. Stosując Let's Encrypt, aby otrzymać certyfikat SSL, należy wcześniej wygenerować parę kluczy prywatny/publiczny, wysłać ją do zaufanego urzędu certyfikacji. Obecnie Let's Encrypt od razu i bezpłatnie generuje oraz waliduje certyfikaty.

Generowanie certyfikatów

Specyfikacja TLS wymaga certyfikatu, który jest podpisywany przez zaufany urząd certyfikacji (CA). CA zapewnia, że posiadacz certyfikatu jest naprawdę tym, za kogo się podaje. Zielona ikona kłódki (lub inny zielonkawy znak po lewej stronie adresu URL w przeglądarce) oznacza, że serwer, z którym się użytkownik komunikuje, jest tym, za kogo się podaje.

Warto zauważyć, że certyfikat ten niekoniecznie musi być zweryfikowany przez organ taki jak Let's Encrypt. Istnieją również inne płatne usługi. Technicznie można podpisać go samemu, ale wtedy (nie będąc zaufanym CA) użytkownicy odwiedzający daną stronę prawdopodobnie zobaczą ostrzeżenie, że strona może być niebezpieczna.

Do generowania i zarządzania certyfikatami Let's Encrypt wymagana jest instalacja Certbota, która dla systemu Windows została opisana na stronie pod adresem [Certbot - Windows Other \(eff.org\)](https://certbot.eff.org/).

Webroot jest wtyczką od Certbota, która oprócz domyślnej funkcjonalności Certbota (automatycznie generuje parę kluczy publiczny/prywatny wraz z certyfikatem SSL), kopiuje certyfikaty do folderu webroot i weryfikuje serwer poprzez umieszczenie kodu weryfikacyjnego w ukrytym katalogu tymczasowym o nazwie *.well-known*. Aby pominąć ręczne wykonywanie niektórych czynności, należy skorzystać z tego pluginu. Plugin jest domyślnie zainstalowany wraz z Certbotem. Do wygenerowania i zweryfikowania certyfikatów, należy wykonać następujące czynności:

```
certbot certonly --webroot -w /var/www/example/ -d www.example.com -d example.com
```

Po uruchomieniu powyższego polecenia na wyjściu pojawiają się ścieżki do plików klucza prywatnego i certyfikatu. Należy skopiować te wartości do zakomentowanego fragmentu kodu.

Listing 9.2 Ścieżki do plików klucza prywatnego i certyfikatu

```
const options = {
  key: fs.readFileSync("/var/www/example/sslcert/privkey.pem"),
  cert: fs.readFileSync("/var/www/example/sslcert/fullchain.pem")
}
```

Zwiększanie bezpieczeństwa poprzez użycie mechanizmu HTTP Strict Transport Security (HSTS), który ma na celu ograniczenie ataków sieciowych typu „protocol downgrade” oraz „cookie hijacking” pokazano na listingu 9.3. W szkielecie Ekspres należy użyć modułu Helmet.



Listing 9.3 Użycie modułu Helmet

```
const https = require("https")
fs = require("fs")
helmet = require("helmet")

const options = {
  key: fs.readFileSync("/srv/www/keys/my-site-key.pem"),
  cert: fs.readFileSync("/srv/www/keys/chain.pem")
}

const app = express()

app.use(helmet()) // oprogramowanie pośredniczące helmet

app.use((req, res) => {
  res.writeHead(200)
  res.end("hello world\n")
})

app.listen(8000)
https.createServer(options, app).listen(8080)
```

Do szyfrowania używane są dwa różne klucze: klucz, który otrzymujemy od urzędu certyfikacji i klucz, który jest generowany przez serwer do wymiany kluczy. Domyślny klucz do wymiany kluczy (nazywany również kluczem Diffie-Hellmana lub DH) używa "mniejszego" klucza niż ten dla certyfikatu. Aby temu zaradzić, zostanie wygenerowany silny klucz DH, a następnie przekazany do naszego bezpiecznego serwera.

Do wygenerowania dłuższego (2048 bitowego) klucza, będzie potrzebny *openssl*. W celu sprawdzenia tego należy uruchomić *openssl -v*. Jeśli polecenie nie zostanie znalezione, należy zainstalować *openssl* za pomocą poniższego polecenia i następnie skopiować ścieżkę do konfiguracji tworzonej aplikacji (Listing 9.4).

```
openssl dhparam -out /var/www/example/sslcrt/dh-strong.pem 2048
```

Listing 9.4 Zastosowanie openssl

```
const options = {
  key: fs.readFileSync("/var/www/example/sslcrt/privkey.pem"),
  cert: fs.readFileSync("/var/www/example/sslcrt/fullchain.pem"), //ścieżki te mogą się różnić od
  dhparam: fs.readFileSync("/var/www/example/sslcrt/dh-strong.pem")
}
```

Zadanie 9.2. Implementacja aplikacji obsługującej formularze z uwzględnieniem aspektów bezpieczeństwa

Obsługa danych pochodzących z formularzy wiąże się z dodatkowymi względami bezpieczeństwa. Opracowywana aplikacja będzie zawierała formularz kontaktowy, za pomocą którego będzie można bezpiecznie wysłać wiadomość i adres email.

Etapy realizacji aplikacji obejmują:

1. Wyświetlenie pustego formularza HTML w odpowiedzi na początkowe żądanie GET
2. Przesłanie przez użytkownika formularza z danymi w żądaniu POST
3. Walidacja po stronie klienta i serwera
4. Wyświetlenie formularza z wypełnionymi danymi i komunikatami o błędach jeśli dane będą niepoprawne
5. Przetwarzanie poprawnych, oczyszczonych danych na serwerze
6. Przekierowanie użytkownika lub wyświetlenie komunikatu o pomyślnym przetworzeniu danych

Listing 9.5 Podstawowy kod serwera aplikacji

```
const path = require('path')
const express = require('express')
const layout = require('express-layout')

const routes = require('./routes')
const app = express()

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'hbs')

const middlewares = [
  layout(),
  express.static(path.join(__dirname, 'public')),
]
app.use(middlewares)

app.use('/', routes)
app.use((req, res, next) => {
  res.status(404).send("Sorry can't find that!")
})

app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
})

app.listen(3000, () => {
  console.log('App running at http://localhost:3000')
})
```

Kod routera *routes.js* zawiera na początku trasę '/', która renderuje widok *index.hbs*.

Listing 9.6 Implementacja routera

```
const express = require('express')
const router = express.Router()

router.get('/', (req, res) => {
  res.render('index')
});

module.exports = router
```

Następnie zostanie wygenerowany formularz na podstawie szablonów: *contact.hbs* oraz *layout.hbs*. W tym momencie należy także dodać trasę */contact* oraz uzupełnić kod serwera o możliwość komunikowania się z szablonami (moduł flash).

Listing 9.7 Dodanie trasy */contact* w pliku *routers.js*

```
router.get('/contact', (req, res) => {
  res.render('contact')
});
```

Listing 9.8 Uzupełniony plik serwera

```
const path = require('path')
const express = require('express')
const session = require('express-session')
const flash = require('express-flash')

const routes = require('./routes')
const app = express()

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'hbs')

const middlewares = [
  express.static(path.join(__dirname, 'public')),
  session({
    secret: 'super-secret-key',
    key: 'super-secret-cookie',
    resave: false,
    saveUninitialized: false,
    cookie: { maxAge: 60000 }
  }),
  flash(),
];
app.use(middlewares)

app.use('/', routes)

app.use((req, res, next) => {
  res.status(404).send("Sorry can't find that!");
})

app.use((err, req, res, next) => {
```



```
console.error(err.stack)
res.status(500).send('Something broke!')
})

app.listen(3000, () => {
  console.log('App running at http://localhost:3000')
})
```

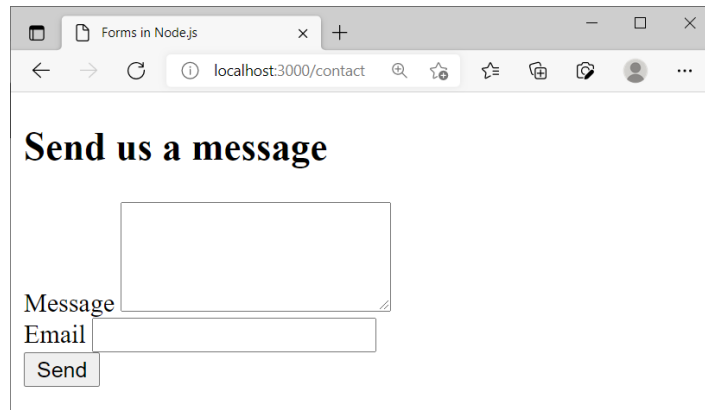
Listing 9.9 *Utworzenie szablonów odpowiedzialnych za wyświetlenie formularza kontaktowego – plik `contact.hbs` oraz `layout.hbs`*

```
<div>
  <h2>Send us a message</h2>
</div>
<form method="post" action="/contact" novalidate>
  <div>
    <label for="message">Message</label>
    <textarea id="message" name="message" rows="4" autofocus></textarea>
  </div>
  <div >
    <label for="email">Email</label>
    <input id="email" name="email" type="email" value="" />
  </div>
  <div >
    <button type="submit">Send</button>
  </div>
</form>
```

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>Forms in Node.js</title>
</head>
<body>
  {{{body}}}
</body>
</html>
```

Efekt powyższego kodu po wprowadzeniu w polu adresu przeglądarki (<http://localhost:3000/contact>) pokazano na Rys. 9.1.



Rys. 9.1 Formularz do wysyłania wiadomości

Po to, aby otrzymywać wartości POST w Expressie trzeba dołączyć do tablicy middleware funkcję pośredniczącą `express.urlencoded({ extended: true })`.

Powszechną konwencją dla formularzy jest to, aby wysyłać dane do tego samego adresu URL, który został użyty w początkowym żądaniu GET. Na Listingu 9.10 zostanie obsłużona trasa `/contact` metodą POST, aby przetworzyć dane wprowadzone przez użytkownika.

Listing 9.10 Obsługa trasy `/contact` metodą GET i POST

```
router.get('/contact', (req, res) => {
  res.render('contact', {
    data: {},
    errors: {}
  })
})
router.post('/contact', (req, res) => {
  res.render('contact', {
    data: req.body, // { message, email }
    errors: {
      message: { msg: 'A message is required' },
      email: { msg: 'That email doesn't look right' }
    }
  })
})
```

Jeśli zgłoszenie jest niepoprawne, należy przekazać z powrotem przesłane wartości do widoku (aby użytkownicy nie musieli ich ponownie wpisywać) wraz z wszelkimi komunikatami o błędach, które trzeba wyświetlić.

W przypadku gdy wystąpią błędy, należy:

- wyświetlić błędy na górze (w nagłówku) formularza
- ustawić wartości wejściowe na takie, jakie zostały przesłane do serwera
- wyświetlić błędy (inline) poniżej lub obok danych wejściowych
- dodać klasę `form-field-invalid` do pól z błędami

Listing 9.11 Obsługa błędów w szablonie formularza




```
<div class="form-header">
  <% if (Object.keys(errors).length === 0) { %>
    <h2>Send us a message</h2>
  <% } else { %>
    <h2 class="errors-heading">Oops, please correct the following:</h2>
    <ul class="errors-list">
      <% Object.values(errors).forEach(error => { %>
        <li><%= error.msg %></li>
      <% } %>
    </ul>
  <% } %>
</div>

<form method="post" action="/contact" novalidate>
  <div class="form-field <%= errors.message ? 'form-field-invalid' : '' %>">
    <label for="message">Message</label>
    <textarea class="input" id="message" name="message" rows="4" autofocus><%= data.message %></textarea>
    <% if (errors.message) { %>
      <div class="error"><%= errors.message.msg %></div>
    <% } %>
  </div>
  <div class="form-field <%= errors.email ? 'form-field-invalid' : '' %>">
    <label for="email">Email</label>
    <input class="input" id="email" name="email" type="email" value="<%= data.email %>" />
    <% if (errors.email) { %>
      <div class="error"><%= errors.email.msg %></div>
    <% } %>
  </div>
  <div class="form-actions">
    <button class="btn" type="submit">Send</button>
  </div>
</form>
```

Walidacja i oczyszczanie danych wejściowych (sanityzacja)

Do tego celu można wykorzystać wygodne middleware o nazwie express-validator przy użyciu biblioteki *validator.js*. Należy dodać ten moduł do aplikacji, co umożliwi następnie walidację i oczyszczenie danych tzn. pozwoli łatwo sprawdzić czy wiadomość oraz adres są poprawne.

Listing 9.12 Kod definicji ścieżek – plik *routes.js* z walidacją pól formularza

```
const { check, validationResult, matchedData } = require('express-validator')

router.post('/contact', [
  check('message')
    .isLength({ min: 1 })
    .withMessage('Message is required'),
  check('email')
```

```
.isEmail()
.withMessage('That email doesn't look right')
], (req, res) => {
  const errors = validationResult(req);
  res.render('contact', {
    data: req.body,
    errors: errors.mapped()
  })
})
```

Oczyszczanie danych

Za pomocą dostarczonych tzw. sanitizatorów, możemy usunąć białe znaki z początku i końca wartości oraz znormalizować adres email do spójnego wzorca. Może to pomóc w usunięciu duplikatów kontaktów utworzonych przez nieco inne dane wejściowe. Na przykład, 'Marek@gmail.com' oraz 'marek@gmail.com' zostaną zamienione na 'marek@gmail.com'.

Sanitizery mogą być dołączone na końcu walidatorów (Listing 9.15).

Listing 9.13 Kod z definicją tras – routes.js, z dodanymi sanitizarami

```
router.post('/contact', [
  check('message')
    .isLength({ min: 1 })
    .withMessage('Message is required')
    .trim(),
  check('email')
    .isEmail()
    .withMessage('That email doesn't look right')
    .bail()
    .trim()
    .normalizeEmail()
], (req, res) => {
  const errors = validationResult(req)
  res.render('contact', {
    data: req.body,
    errors: errors.mapped()
  })

  const data = matchedData(req)
  console.log('Sanitized:', data)
})
```

Funkcja *matchedData()* zwraca wynik działania sanitizatorów na danych wejściowych.

Użycie metody *bail*, która zatrzymuje walidację, jeśli którakolwiek z poprzednich walidacji zakończyła się niepowodzeniem jest potrzebne, ponieważ jeśli użytkownik prześle formularz bez wpisania wartości w polu email, *normalizeEmail* spróbuje znormalizować pusty ciąg znaków i przekonwertować go na @. Zostanie on następnie wstawiony do pola email, gdy ponownie zostanie wyrenderowany formularz.

Walidacja formularza

Jeśli wystąpiły błędy, należy ponownie wyrenderować widok. Jeśli nie, trzeba zrobić coś z danymi, a następnie pokazać, że wysłanie formularza zakończyło się sukcesem. Zazwyczaj użytkownik jest przekierowywany na stronę z informacją o powodzeniu wysłania wiadomości.

Ponieważ protokół HTTP jest bezstanowy, więc nie można przekierować się na inną stronę i przekazać wiadomość bez pomocy ciasteczka sesyjnego, które utrzymuje wiadomość pomiędzy żądaniami HTTP. Komunikat "flash" to nazwa nadana temu rodzajowi jednorazowej wiadomości, która ma być utrzymywana przez przekierowanie, a następnie ma zniknąć.

Istnieją trzy programy pośredniczące, które trzeba dołączyć: cookie-parser, express-session, express-flash (Listing 9.14).

Listing 9.14 Dołączanie middleware do aplikacji – plik *server.js*

```
const cookieParser = require('cookie-parser')
const session = require('express-session')
const flash = require('express-flash')

const middlewares = [
  // ...
  cookieParser(),
  session({
    secret: 'super-secret-key',
    key: 'super-secret-cookie',
    resave: false,
    saveUninitialized: false,
    cookie: { maxAge: 60000 }
  }),
  flash(),
]
```

Dzięki modułowi express-flash dodawany jest *req.flash(type, message)*, który można wykorzystać przy obsłudze tras (Listing 9.15).

Listing 9.15 Obsługa tras z wykorzystaniem możliwości

```
router.post('/contact', [
  // validation ...
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.render('contact', {
      data: req.body,
      errors: errors.mapped()
    })
  }
})
```



```
const data = matchedData(req)
console.log('Sanitized: ', data)

req.flash('success', 'Thanks for the message! I'll be in touch :)')
res.redirect('/')
})
```

Oprogramowanie pośredniczące express-flash dodaje wiadomości do *req.locals*, do których wszystkie widoki mają dostęp (Listing 9.16).

Listing 9.16 Kod widoku *index.hbs*

```
<% if (messages.success) { %>
  <div class="flash flash-success"><%= messages.success %></div>
<% } %>
<h1>Working With Forms in Node.js</h1>
```

Aspekty bezpieczeństwa aplikacji

Praca z formularzami i sesjami w Internecie, wymaga świadomości powszechnych luk w zabezpieczeniach aplikacji internetowych.

TLS vs HTTPS

Zawsze należy używać szyfrowania TLS poprzez `https://` podczas pracy z formularzami, aby przesłane dane były zaszyfrowane podczas przesyłania ich przez Internet. Jeśli wysyłane są dane formularzy przez `http://`, są one wysyłane w postaci zwykłego tekstu i mogą być widoczne dla każdego, kto podsłuchuje te pakiety podczas ich przesyłania przez Internet.

Istnieje małe middleware o nazwie *helmet*, które zwiększa bezpieczeństwo nagłówków HTTP. Najlepiej dołączyć go na samej górze middleware'u. *Helmet* jest bardzo łatwy do włączenia (Listing 9.17).

Listing 9.17 Dołączanie oprogramowania *helmet*

```
const helmet = require('helmet')
middlewares = [
  helmet(),
  // ...
]
```

Cross-site Request Forgery (CSRF)

Zabezpieczenie się przed fałszowaniem żądań przesyłanych między witrynami – atak Cross-Site Request Forgery (CSRF) jest możliwe poprzez wygenerowanie unikalnego tokena, w momencie kiedy użytkownik otrzyma formularz, a następnie weryfikuje token przed przetworzeniem danych POST. Istnieje oprogramowanie pośredniczące, które pomoże się zabezpieczyć przed taką sytuacją.

Listing 9.18 Oprogramowanie pośredniczące dostarczające zabezpieczenia się przed atakami typu *CSRF* – plik *routes.js*



```
const csrf = require('csrf')
const csrfProtection = csrf({ cookie: true })
```

W żądaniu GET generuje się token (Listing 9.19). Również w reakcji na błędy po sprawdzeniu pól formularza w odpowiedzi generowany jest token (Listing 9.20).

Listing 9.19 Generowanie tokena

```
router.get('/contact', csrfProtection, (req, res) => {
  res.render('contact', {
    data: {},
    errors: {},
    csrfToken: req.csrfToken()
  })
})
```

Listing 9.20 Odpowiedź w reakcji na błędy danych w formularzu zawierająca token

```
router.post('/contact', csrfProtection, [
  // validations ...
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.render('contact', {
      data: req.body,
      errors: errors.mapped(),
      csrfToken: req.csrfToken()
    });
  }
  // ...
})
```

Następnie należy dołączyć token w ukrytym komponencie formularza typu input (Listing 9.21).

Listing 9.21 Widok *contact.hbs* z ukrytym komponentem

```
<form method="post" action="/contact" novalidate>
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  <!-- ... -->
</form>
```

Nie trzeba teraz modyfikować modułu obsługi żądań POST, ponieważ wszystkie żądania POST będą teraz wymagały prawidłowego tokenu przez oprogramowanie pośredniczące csrf. Jeśli nie zostanie dostarczony prawidłowy token CSRF, zostanie zgłoszony błąd *ForbiddenError*, który może być obsługiwany przez program obsługi błędów zdefiniowany na końcu *serwera.js*.

Cross-Site Scripting (XSS)

Podczas wyświetlania danych przesyłanych przez użytkownika w widoku HTML należy zachować ostrożność, ponieważ może to otworzyć aplikację na skrypty krzyżowe (XSS).



Wszystkie języki szablonów udostępniają różne metody wyprowadzania wartości. EJS `<%= wartość %>` wyprowadza wartość kodów HTML, aby chronić użytkownika przed podatnością XSS, podczas gdy `<% - wartość %>` wyprowadza nieprzetworzony ciąg.

Zawsze należy używać wyjścia ze znakiem ucieczki `<% = wartość %>` podczas pracy z wartościami przesłanymi przez użytkownika. Surowych danych wyjściowych należy używać tylko wtedy, gdy jest pewność, że jest to bezpieczne.

Dołączanie do wiadomości i przesyłanie plików

Przesyłanie plików w formularzach HTML jest szczególnym przypadkiem, który wymaga kodowania typu "multipart/form-data". Do tego będzie potrzebne dodatkowe oprogramowanie pośredniczące do obsługi przesyłania wieloczęściowego. W Expressie można wykorzystać pakiet o nazwie `multer` (Listing 9.22). W tym kodzie wysyłany jest do pamięci plik poprzez pole `photo` za pomocą `req.file`.

Listing 9.22 Użycie pakietu `multer` do przesyłania z formularzem plików – `routes.js`

```
const multer = require('multer')
const upload = multer({ storage: multer.memoryStorage() })

router.post('/contact', upload.single('photo'), csrfProtection, [
  // validation ...
], (req, res) => {
  // error handling ...

  if (req.file) {
    console.log('Uploaded: ', req.file)
    // Homework: Upload file to S3
  }

  req.flash('success', 'Thanks for the message! I'll be in touch :)')
  res.redirect('/')
})
```

Ostatnią rzeczą będzie dodanie atrybutu `enctype` do pliku wejściowego (Listing 9.23).

Listing 9.23 Dodanie atrybutu `enctype` do formularza

```
<form method="post" action="/contact?_csrf=<%= csrfToken %>" novalidate
enctype="multipart/form-data">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  <div class="form-field <%= errors.message ? 'form-field-invalid' : '' %>">
    <label for="message">Message</label>
    <textarea class="input" id="message" name="message" rows="4" autofocus><%= data.message
%></textarea>
    <% if (errors.message) { %>
      <div class="error"><%= errors.message.msg %></div>
    <% } %>
  </div>
  <div class="form-field <%= errors.email ? 'form-field-invalid' : '' %>">
    <label for="email">Email</label>
    <input class="input" id="email" name="email" type="email" value="<%= data.email %>" />
    <% if (errors.email) { %>
```



```
<div class="error"><%= errors.email.msg %></div>
<% } %>
</div>
<div class="form-field">
  <label for="photo">Photo</label>
  <input class="input" id="photo" name="photo" type="file" />
</div>
<div class="form-actions">
  <button class="btn" type="submit">Send</button>
</div>
</form>
```

Zadanie 9.3 Rozbudowa aplikacji przesyłających dane

- a) Rozszerz program o wysyłanie danych z formularza przy pomocy poczty internetowej.
- b) Rozszerz funkcjonalność aplikacji poprzez zapis danych z formularza w bazie danych.
- c) Zamiast szablonów handlebars użyj biblioteki React.



Materiały zostały opracowane w ramach projektu
„*Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga*”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego