Wed Development SE371-IS311

Chapter 8

Managing State

Sessions, cookies, authentication

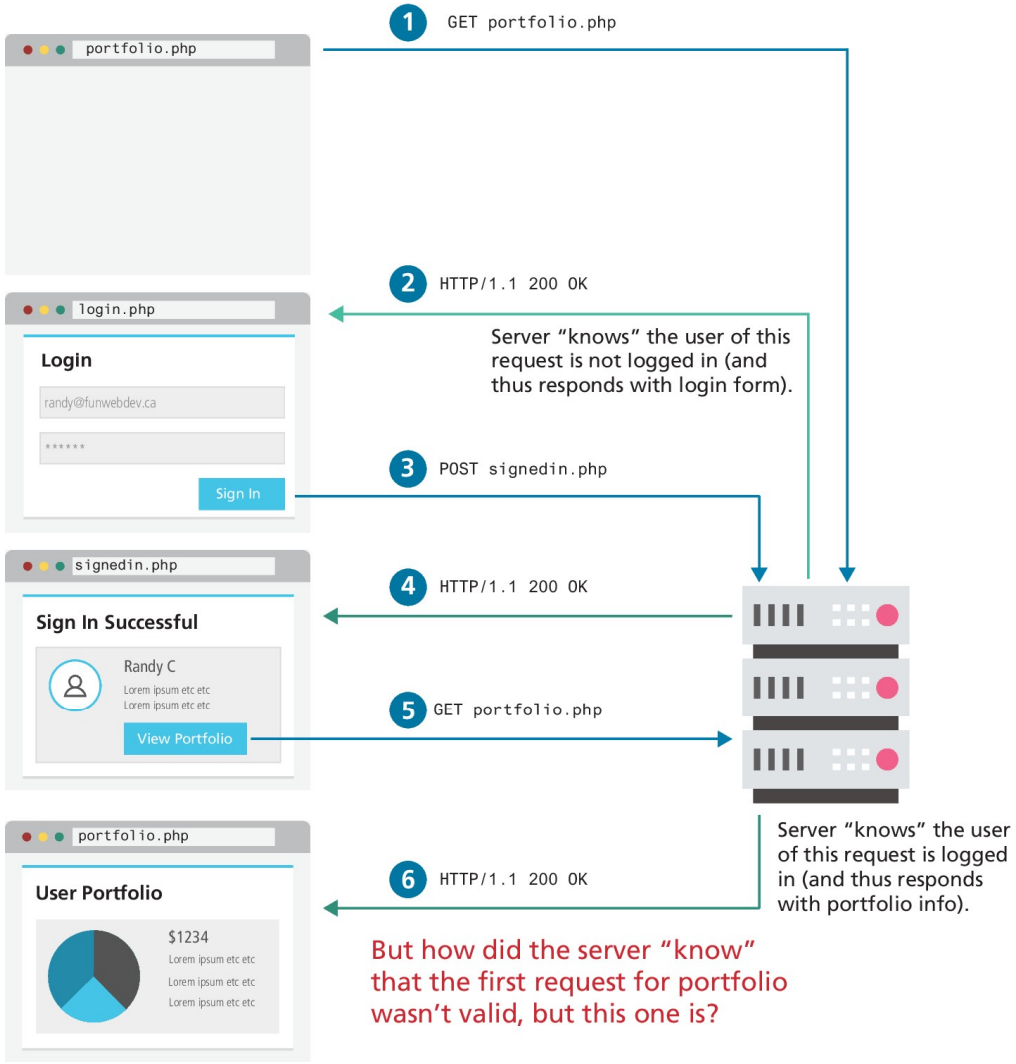
Topics covered

- Why state is a problem in web application development
- What cookies are and how to use them
- What session state is and what are its typical uses and limitations
- How to implement authentication

The Problem of State in Web Applications

How can one request share information with another request?

The question is: how can the server "know" if the user was or was not logged in? →

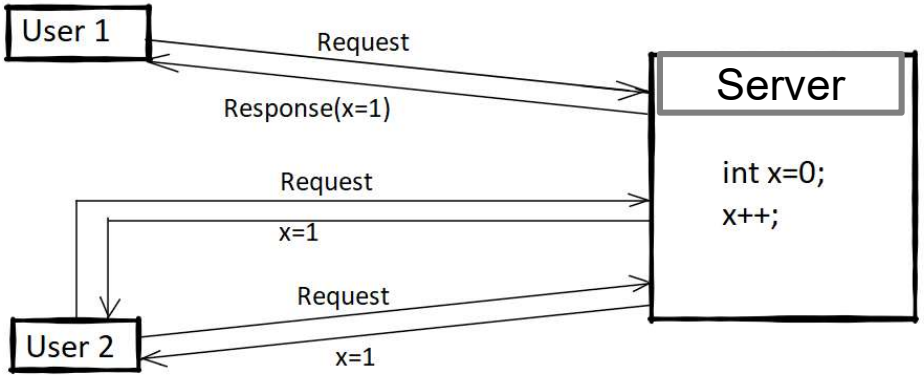


The Problem of State in Web Applications (ii)

Unlike a desktop application, A web application consists of a series of **independent** HTTP requests.

HTTP is stateless, but not sessionless: There is no link between two HTTP requests being successively carried out on the same connection. This is a problem for users attempting to complete a task with different pages to access or different actions to perform: for example, using e-commerce shopping cart.

But HTTP cookies allow the use of stateful sessions. Using header extensibility, HTTP Cookies are added to the workflow, allowing to share the same state between HTTP requests.



Cookies

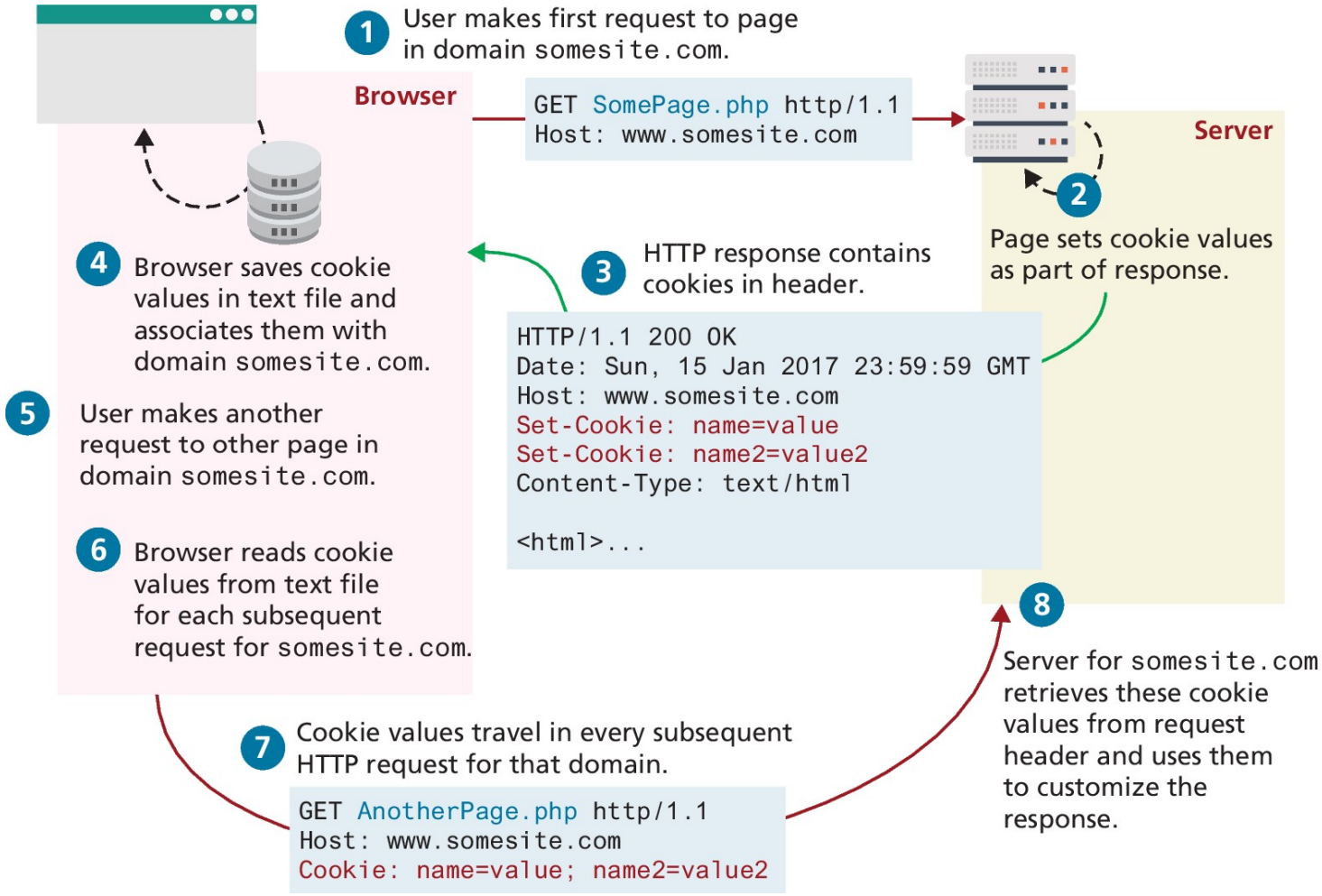
Cookies are a client-side approach for persisting state information.

They are name=value pairs that are saved within one or more text files that are managed by the browser.

While cookies can be used for any state-related purpose, they are principally used as a way of maintaining continuity over time in a web application. One typical use of cookies in a website is to “remember” the identity of the visitor so that the server can customize the site for him.

Cookies are also frequently used to keep track of whether a user has logged into a site.

How Do Cookies Work?



Using Cookies in Node and Express

Cookie support in Node and Express (cookie-parser) needs to be installed using npm.

httpOnly: when set to true, the cookie is not readable by JS.

request.cookies: contains received cookie key/value pairs.

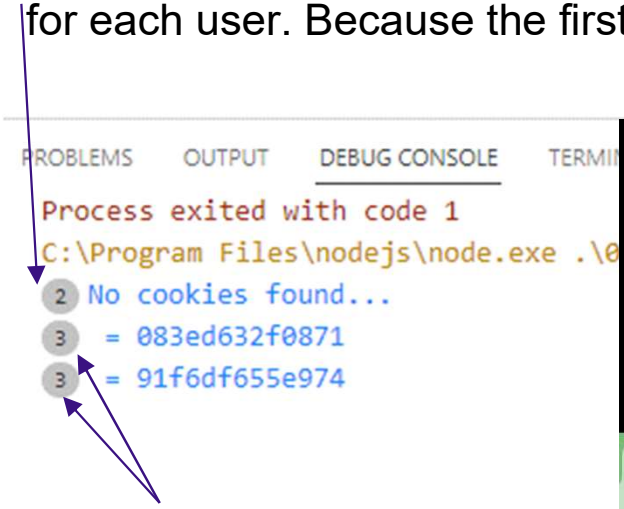
```
const express = require('express');
const app = express();
const cookieParser = require('cookie-parser');

app.use(cookieParser());

app.get('/', (req, resp) => {
  const opts = {
    maxAge: 24 * 60 * 60 * 1000, // set age limit to 1 day
    httpOnly: true
  }
  const entries = Object.entries(req.cookies); //transforms an object to an
                                               // array of key-value pairs
  if (entries.length == 0) {
    console.log(`No cookies found, generating new ID for user...`);
    // now write new cookie with new ID as part of response
    resp.cookie('user', Math.random().toString(16).slice(2), opts);
  } else {
    for (const [name, value] of entries) {// loop through all cookies
      // resend same cookies found in request
      resp.cookie(`${name}`, `${value}`, opts);
    }
  }
  resp.send('content sent to browser');
});

app.listen(3000);
```

Source code: chapter08/01_cookies/

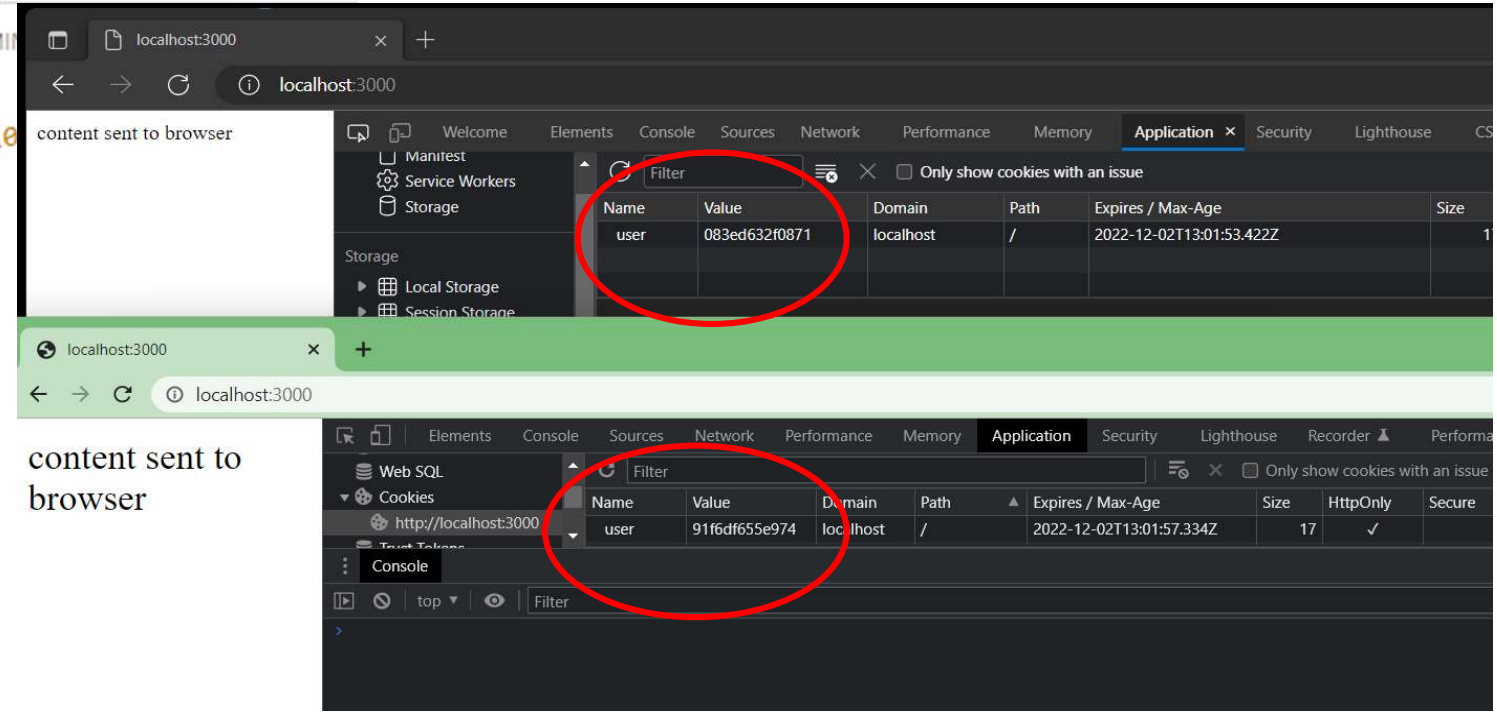
Using Cookies in Node and Express (2)

Now, we can test the previous code by sending HTTP requests from two different browsers:

We can see in the server terminal that the “No cookies found…” message was logged two times, one for each user. Because the first time any user sends a request, he sends no cookies in the request.



Then, any of both users is sending his cookie with his assigned ID. Here, we sent four HTTP requests from each client.



Persistent Cookie Best Practices

Due to the limitations of cookies (small size, vulnerable to attacks) your site's correct operation should not be dependent upon them.

Almost all login systems are dependent upon IDs sent in session cookies

Cookies containing sensitive information should have a short lifetime

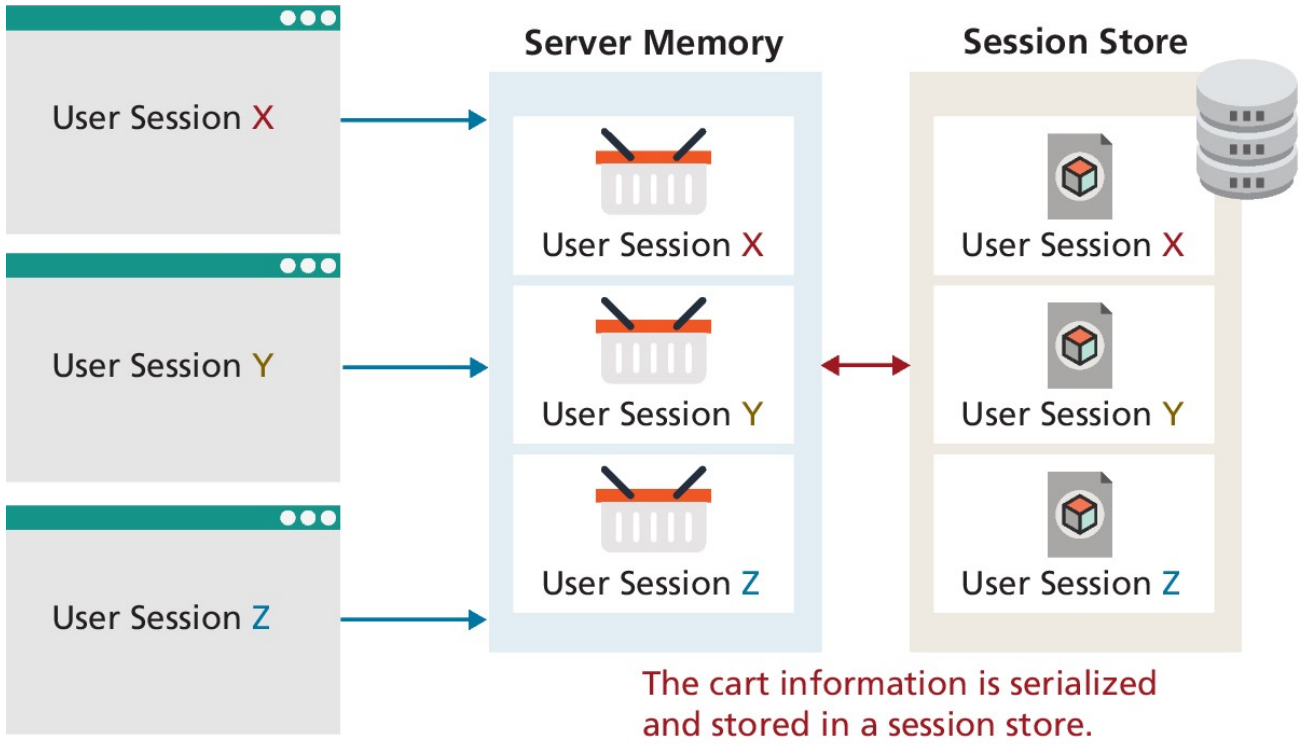
A login cookie might contain the username but not the password. Instead, the login cookie would contain a random token used by the site's back-end database. Every time the user logs in, a new token would be generated and stored in the database and sent to the user as cookie.

Google recommends that cookies occupy no more than 400 bytes (or 200 characters) to maximize performance. They also recommend that static files such as images, CSS, and JavaScript come from a distinct domain with cookies disabled.

Session State

Session state is a server-based state mechanism that lets web applications store and retrieve objects of any type for each unique user session.

Session state is dependent upon some type of **session store**, that is, some type of storage area for session information.

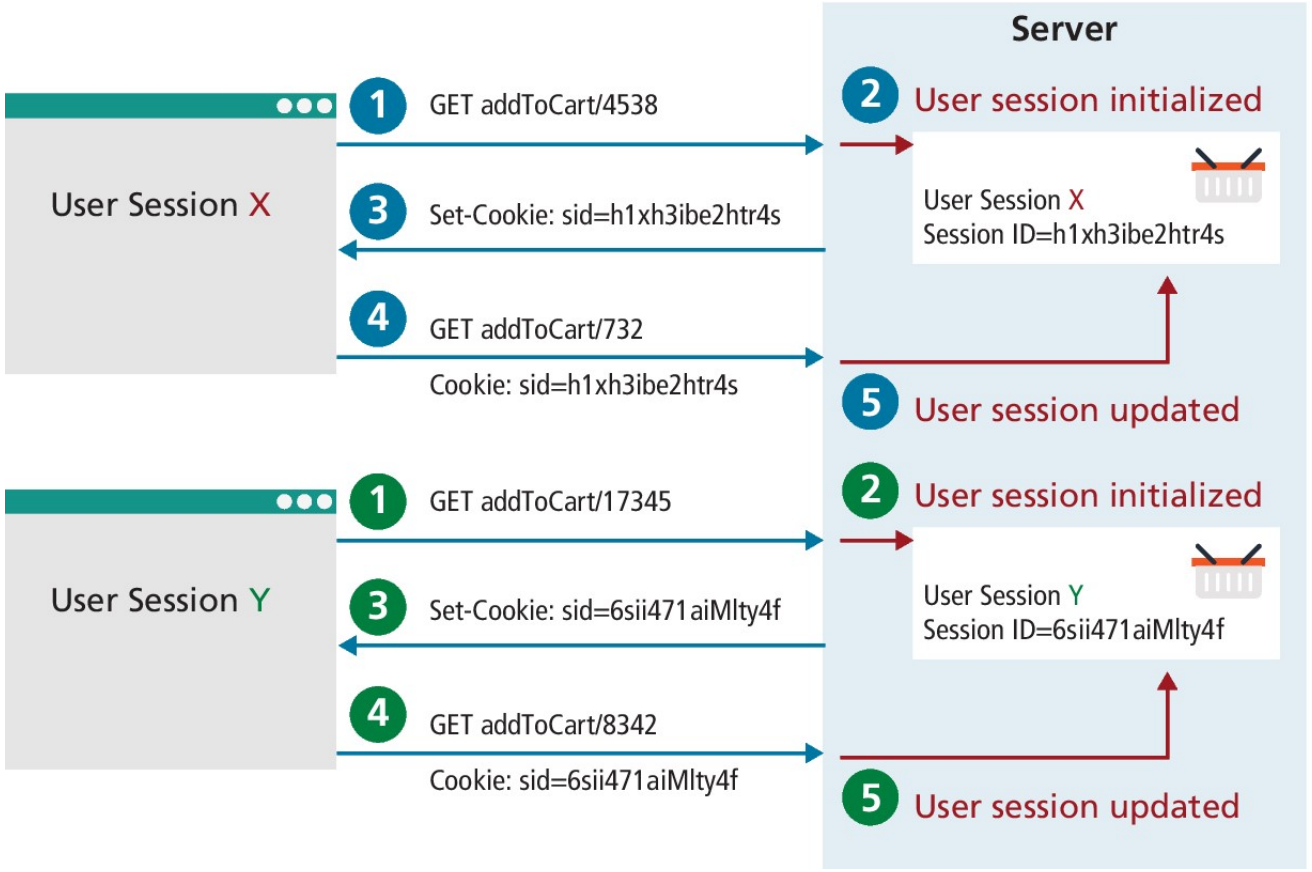


How Does Session State Work?

Since HTTP is stateless, some type of user/session identification system is needed.

Sessions in Express are identified with a unique session ID.

This session ID is transmitted back and forth between the user and the server via a session cookie



Session State in Node

Source code: chapter08/02_sessions_intro/

The **express-session** package supports different session stores, whether they be memory, files, external caches, or databases.

This example demonstrates how the express-session package could implement a simple favorites list.

SESSION_SECRET is used by express-session to encrypt the sessionId

```
const express = require('express');
const session = require('express-session');
require('dotenv').config();
const app = express();

// configure session middleware
app.use(session({
  secret: process.env.SESSION_SECRET,
  saveUninitialized: false,
  resave: false
}));

app.get('/addToFavorites/:prodid', function(req, resp) {
  if (req.session.favorites) {
    const favorites = req.session.favorites;
    favorites.push( req.params.prodid );
  } else {
    req.session.favorites = [ req.params.prodid ];
  }
  resp.send('content sent to browser');
});

app.listen(3000);
```

Session store mechanisms

The default session store mechanism in express is **server memory**, which isn't suitable for production environments as the data is lost if the server crashes and had to restart, also when you have thousands of simultaneous connections the size of session objects in memory becomes too large.

In production, we always use a database store to persist sessions:

There are dozens of compatible session store packages that allow you to use a wide range of databases and cloud services for your session store.

‘connect-mongodb-session’ is one of the most used one within express.

Store sessions in MongoDB with ‘connect-mongodb-session’

```
const express = require('express');
const session = require('express-session');
const MongoDBStore = require('connect-mongodb-session')(session);
require('dotenv').config();

const app = express();

let sessionStore = new MongoDBStore({
  uri: process.env.MONGO_URI,
  collection: 'mySessions'
});

// Catch errors
sessionStore.on('error', function(error) {
  console.log(error);
});

// configure session middleware
app.use(session({
  secret: process.env.SESSION_SECRET,
  cookie: {
    maxAge: 1000 * 60 * 60 * 24 * 7, // 1 week
  },
  saveUninitialized: true,
  resave: true,
  store: sessionStore
}));
```

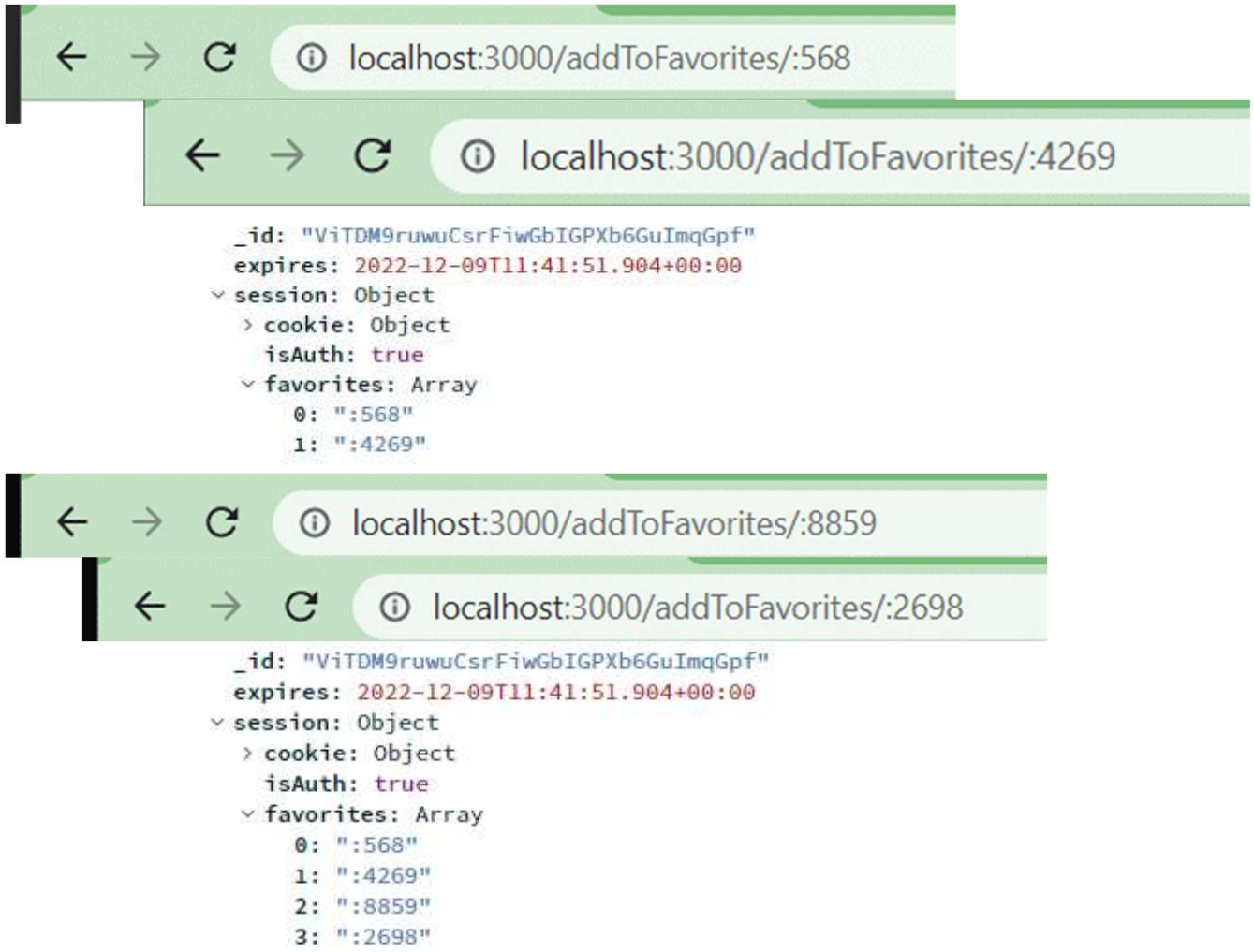
```
app.get('/addToFavorites/:prodid', function(req, resp) {
  console.log(req.session);
  req.session.isAuth = true;
  if (req.session.favorites) {
    const favorites = req.session.favorites;
    favorites.push( req.params.prodid );
  } else {
    req.session.favorites = [ req.params.prodid ];
  }
  // send message or do something else
  req.session.save(
    () => {resp.send('content sent to browser');}
  );

});


app.listen(process.env.port, 'localhost', () => {
    console.log(`Listening on port ${process.env.PORT}`);
});
```

Source code: chapter08/03_stored_sessions/

MongoDB as a session store: (testing)

1. To test the previous code, we start the server then send requests to add some products to the favorites list of the client.

2. Then we re-start the server and send other requests:



Authentication: An introduction

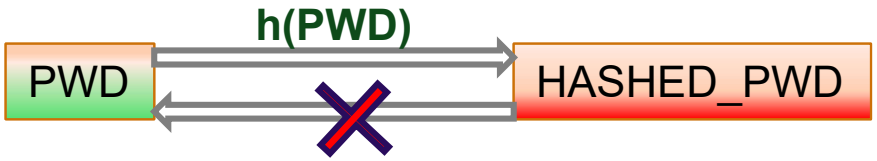
- Authentication and authorization are different concepts:
  - Authentication: confirm a user identify (for example by receiving his username and password and validating them). (HTTP error: 401 Unauthorized)
  - Authorization: Grant an authenticated user access to resources/services etc. (HTTP error: 403 Forbidden).
- There are different methods of authentication:
  - Session-based
  - Token-based
  - Passwordless

Authentication: An introduction (2)

- When a user wants to execute any operation for which we need to make sure about his identity, we use authentication. Authentication usually uses passwords.
- The password should never be stored anywhere in clear, otherwise, if the security of the server is compromised, the passwords can be stolen.
- We use hashing, then we only store the hash of passwords. We apply a hashing function to the inmput: MD5, SHA-256, bcrypt, etc.
- Example of SHA-256 hash: Input: Selemou alykom people

-> output: db863c335ceb3db9a75a9872158326e5ebf68b667f51379f27342a3d1750eb6e

The main property of hashing is **irreversibility**: If a hacker has the hashed password, he cannot (in a reasonable amount of time) compute the original password even if he knows the hashing algorithm used.



Authentication: An introduction (3)

- Session-based: users' credentials (username/email and password for example) are compared with what is stored in the database and if they match, a session is initialized for the user with the fetched id. These sessions are stored in the server and terminated on user logout, they also expire after a configured time.

- Token-based: users' credentials are compared with what is stored in the database and if they match, a Token is initialized and sent to the user. The token is only stored at the client side.

- Passwordless: allows users to log in without the need to remember a password. Instead, users enter their mobile phone number or email address and receive a one-time code or link, which they can use to log in.

Session-based authentication

- The first step for a user is to register to our website.
- First, the user sends a POST request from a registration form. We check if the email is already in our DB.
- If no, we hash the password and store the email, username and the hashed password in the DB.

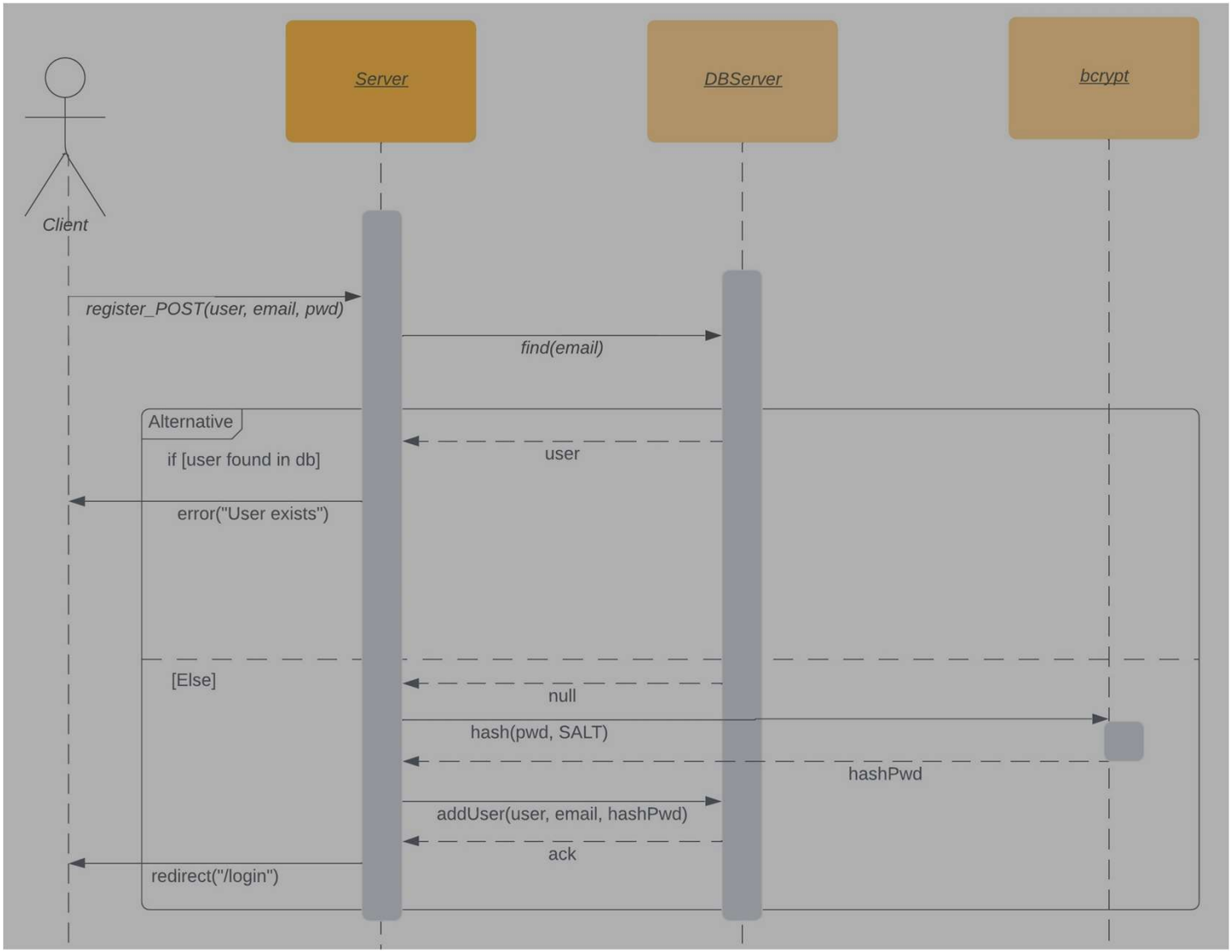


fig. Registration process.

Session-based authentication (2)

- The second step for a user is to login to our website.
- We first verify that the user exists in the db. If yes, we compare the pwd he entered with the hashPWD that is stored in the DB using bcrypt.
- If the credentials are correct, we add to the session of the user his username and a boolean isAuth = true.

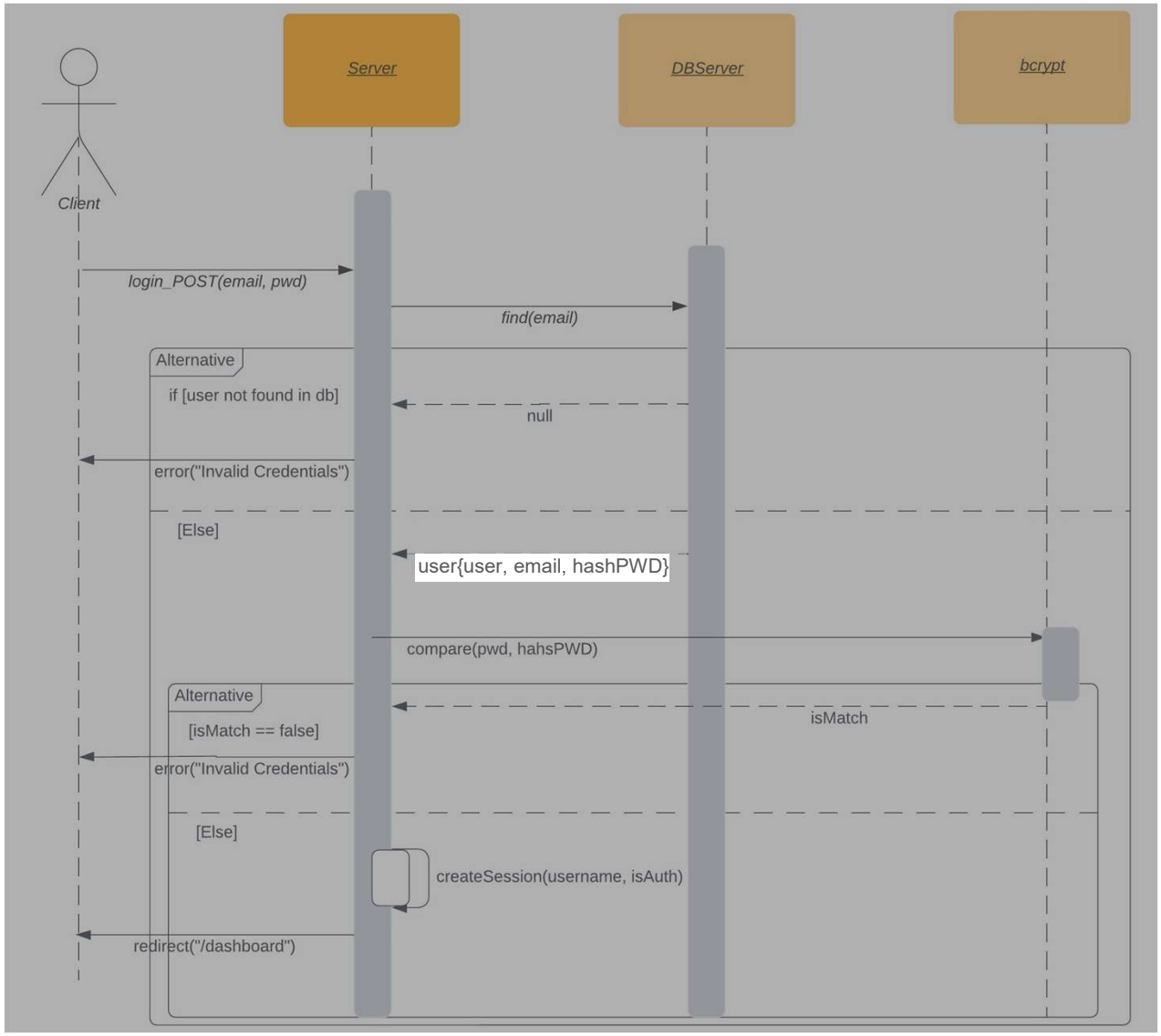


fig. Login process.

Session-based authentication (3)

/register page with a registration form

```
<div class="form">
  <% if(err) { %>
  <p><%= err %></p>
  <% } %>
  <h3>Register Page</h3>
  <form action="/register" method="POST">
    <div class="input-group">
      <label for="username">Username:</label>
      <input type="text" name="username" required/>
    </div>
    <div class="input-group">
      <label for="email">Email:</label>
      <input type="email" name="email" required/>
    </div>
    <div class="input-group">
      <label for="password">Password:</label>
      <input type="password" name="password" required/>
    </div>
    <button>Register</button>
  </form>
  <a href="/login">Login</a>
</div>
```

/login page with a login form

```
<div class="form">
  <% if(err) { %>
  <p><%= err %></p>
  <% } %>
  <h3>Login Page</h3>
  <form action="/login" method="POST">
    <div class="input-group">
      <label for="email">Email:</label>
      <input type="email" name="email" />
    </div>
    <div class="input-group">
      <label for="password">Password:</label>
      <input type="password" name="password" />
    </div>
    <button>Login</button>
  </form>
  <a href="/register">Register</a>
</div>
```

Source code: chapter08/04_authentication/

Session-based authentication (4)

App.js

```
const express = require('express');
const session = require('express-session');
const mongoose = require('mongoose');
const MongoDBStore = require('connect-mongodb-session')(session);
require('dotenv').config();

const appController = require("./controllers/appController");
const isAuth = require("./middleware/isAuth");

const app = express();
app.set("view engine", "ejs");
app.use(express.static('public'));
app.use(express.urlencoded({ extended: true }));

let sessionStore = new MongoDBStore({
  uri: process.env.MONGO_URI,
  collection: 'mySessions'
});

// Catch errors
sessionStore.on('error', function(error) {
  console.log(error);
});
// configure session middleware
app.use(session({
  secret: process.env.SESSION_SECRET,
  cookie: {
    maxAge: 1000 * 60 * 60 * 24 * 7, // 1 week
  },
  saveUninitialized: true,
  resave: true,
  store: sessionStore
}));
```

```
// Landing Page
app.get("/", appController.landing_page);

// Login Page
app.get("/login", appController.login_get);
app.post("/login", appController.login_post);

// Register Page
app.get("/register", appController.register_get);
app.post("/register", appController.register_post);

// Dashboard Page
app.get("/dashboard", isAuth, appController.dashboard_get);

app.post("/logout", appController.logout_post);

// Connect to database then start server
mongoose.connect(process.env.MONGO_URI)
  .then((result) => {
    console.log("Connected to database...");
    app.listen(process.env.port, 'localhost', () => {
      console.log(`Listening on port ${process.env.PORT}`);
    });
  })
  .catch((err) => {
    console.log(err);
  });
```

Session-based authentication (4)

/controllers/appController.js

```
const bcrypt = require("bcryptjs");

const User = require("../models/User");

const landing_page = (req, res) => {
  res.render("landing");
};

const login_get = (req, res) => {
  const error = req.session.error;
  delete req.session.error;
  res.render("login", { err: error });
};
```

```
const login_post = async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ email });

  if (!user) {
    req.session.error = "Invalid Credentials";
    return res.redirect("/login");
  }

  const isMatch = await bcrypt.compare(password,
user.password);

  if (!isMatch) {
    req.session.error = "Invalid Credentials";
    return res.redirect("/login");
  }

  req.session.isAuth = true;
  req.session.username = user.username;
  res.redirect("/dashboard");
};
```

Session-based authentication (5)

/controllers/appController.js (2)

```
const register_get = (req, res) => {
  const error = req.session.error;
  req.session.error = undefined;
  res.render("register", { err: error });
};

const register_post = async (req, res) => {
  const { username, email, password } = req.body;

  let user = await User.findOne({ email: email });
  if (user) {
    req.session.error = "User already exists";
    return res.redirect("/register");
  }
  const hasdPsw = await bcrypt.hash(password, 12);

  user = new User({
    username,
    email,
    password: hasdPsw,
  });

  await user.save();
  res.redirect("/login");
};
```

```
const dashboard_get = (req, res) => {
  const username = req.session.username;
  res.render("dashboard", { name: username });
};

const logout_post = (req, res) => {
  req.session.destroy((err) => {
    if (err) throw err;
    res.redirect("/login");
  });
};

module.exports = { landing_page, login_get, login_post,
                   register_get, register_post,
                  dashboard_get, logout_post };
```

END