The Hackerman Cometh

Introduction

This project will inform users about computer security exploits through interactive tutorials. The goal of this project is to assist another generation to white hat hack and thus, lead to another generation of ethical hackers. Additionally, we aim to teach web developers about security so that they don't write insecure code that leads to harming their product and or users. So, by nature our audience does have a technical background and are seeking to further that knowledge by learning more about computer security. The constraints we face are a 5/04/20 deadline, the risks pertaining to our team not having in depth experience with security exploits and the technologies used in the project, the Agile methodology we work under (Scrumban), and making sure we don't go out of scope by conferring with our stakeholders (the members of our team and Dr. V).

Project Glossary

Term	**Definition**
Hackerman	The user that attempts to hack our system.
Tutorial	A resource that teaches the user about a given security exploit.
Dialogue Box	Text box that pops up containing relevant information on the current task of the tutorial or other hints the user might need.
Hack	The goal(s) needed to accomplish a tutorial.
Injection	Using code that is native to the language in question, and manipulating the mother language to extract information.

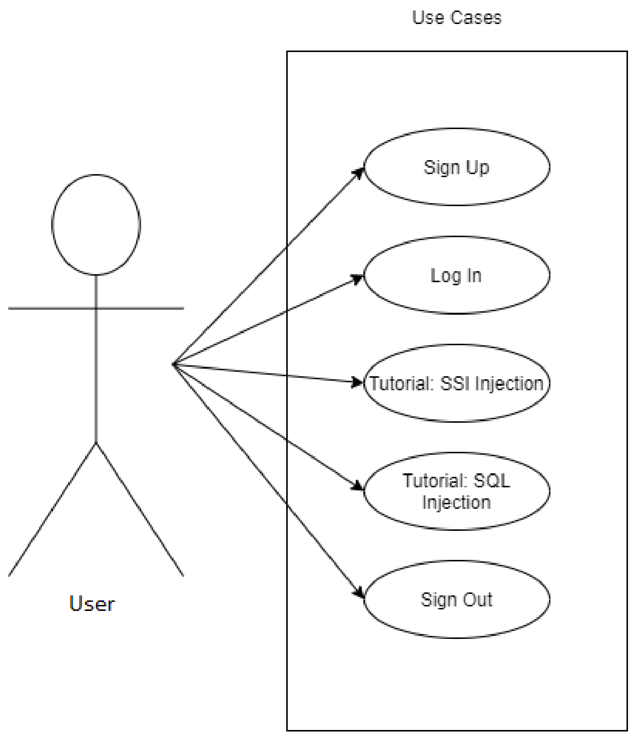
Inspect Element	Browser tutorial tool that allows you to look at the source code for any given web page.
SQL	Structured Query Language, a language used to query a database.
SQL Injection	A method of code manipulation that allows a knowledgeable hacker to manipulate the data found in an SQL server. The hacker inputs an SQL query that will always return true i.e. 1=1 and thusly results in accessing the database where you shouldn't be able to.
File Manipulation	Changing the contents of a file, often so that the manipulator gains an advantage for something.
SSI	Server Side Includes is a simple interpreted language for server-side scripting. It is used for server operations across the World Wide Web and is commonly found on Apache servers.
SSI Injection	A method of code injection using SSI, typically this method is used when a known output is generated from a text box. The user can inject SSI scripts and the results are displayed in the output.
HTML	Hypertext Markup Language, typically used to display content in a web browser.
Security Exploit	An unintentional and or unpatched fault that allows for unintended effects in the software system.
URL	URL (Uniform Resource Locator), also known as a web address indicates the location of a web page.

SPA	A SPA (single-page application), changes the content of a web page dynamically, allowing for a change in content that does not cause a page refresh.
Objective	An introduction to a tutorial, along with a description of what the tutorial will accomplish.
White hat	An ethical hacker.
GUI	Graphical User Interface, the portion of the code that is seen by the user.
Frontend	Analogues for frontend are GUI and client-side.
Backend	An analogue for backend is server or server-side.
SPA	A single-page application (SPA), changes the content of a web page dynamically, allowing for a change in content that does not cause a page refresh.
HTML	Hypertext Markup Language, typically used to display content in a web browser.
Component library	A library that provides styling and common components that aren’t immediately available from the HTML spec. A component library allows for us to more quickly build GUIs and provide consistency in our styling across components.
React	React is a SPA library for building interactive GUIs.
Ant Design	Ant Design in our case, is a component library for React.

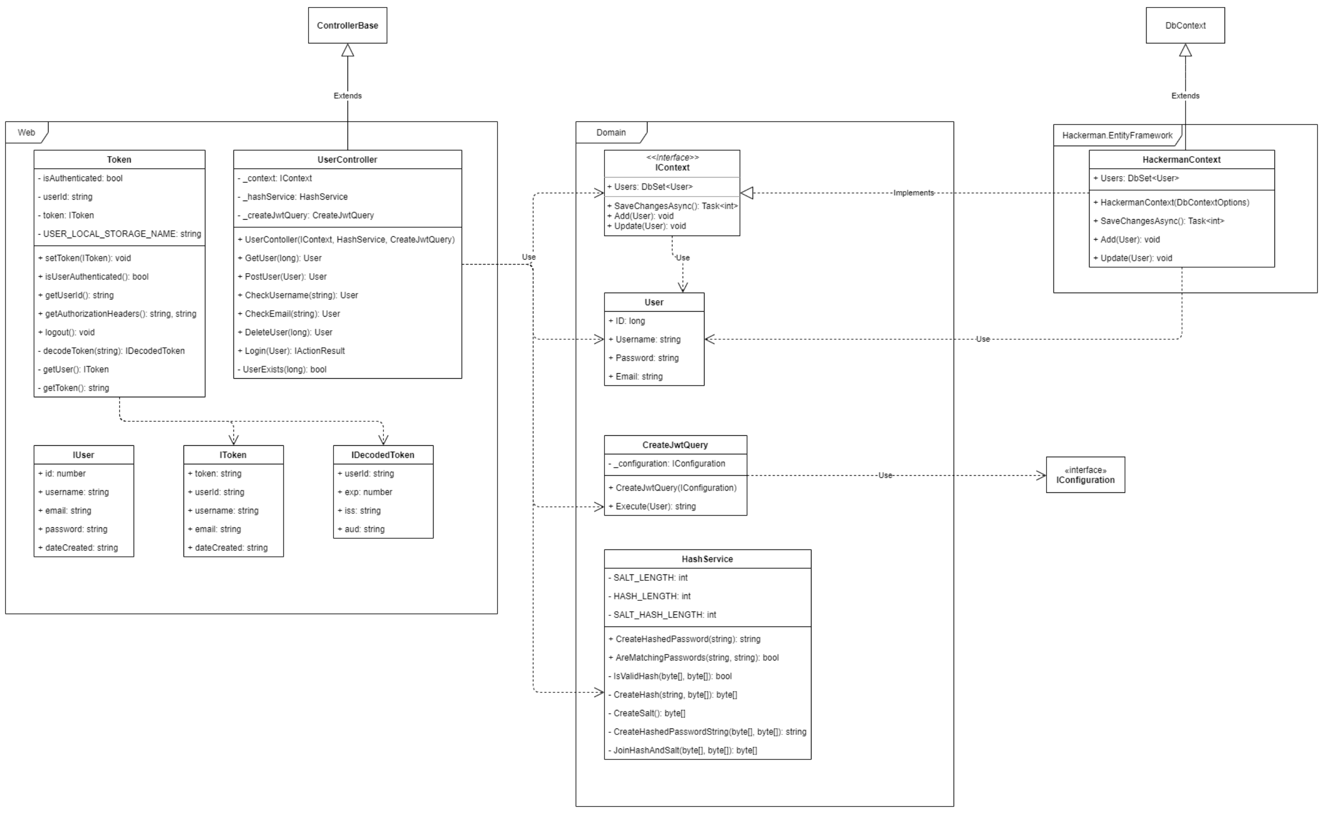
Gatsby	Gatsby is a static-site generator for React that has a large emphasis on performance and maintainability.
Finite State Machine	In our case, it’s a mechanism that explicitly expresses the state of a GUI based on actions. This mechanism will be used in the tutorials, to easily determine what state the user is in and protect against accidental states and bugs that many booleans could cause.
JWT	JSON Web Tokens (JWTs) are our method for maintaining secure authentication for a user. This security method is often used in SPAs because it’s excessive to ping the server every time a user changes pages.
AJAX	Asynchronous JavaScript and XML (AJAX) allows us to send/get data to/from the server without refreshing the page.
REST	Representational state transfer (REST) allows us to set up endpoints on the backend to provide the frontend with data, typically from the database via AJAX calls.
ASP.NET Core	ASP.NET Core is a backend framework for C#. The main feature it provides for us is easy REST API endpoint creation.
Entity Framework Core	Entity Framework Core is an object relational mapper (ORM), which allows us to interact with our database in a more maintainable and intuitive way via C# abstractions.

Popover	UI element that points to another UI element and has 2 parts. The part above the line is the action that the user is performing and that action also relates to where it’s pointing. The part below the line is the description of why that action is being performed and any other information that would be helpful for learning purposes.

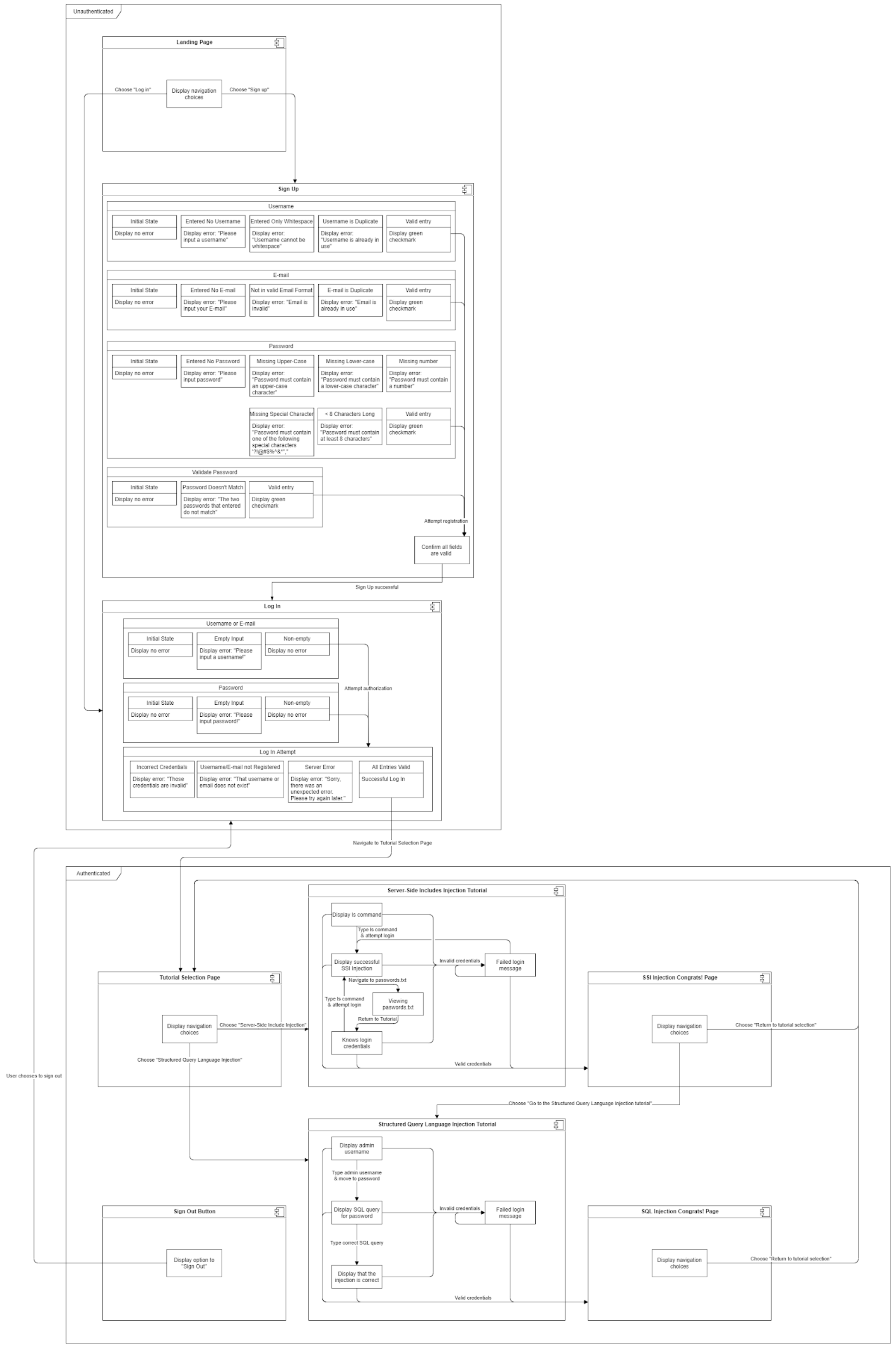
Use Case Model



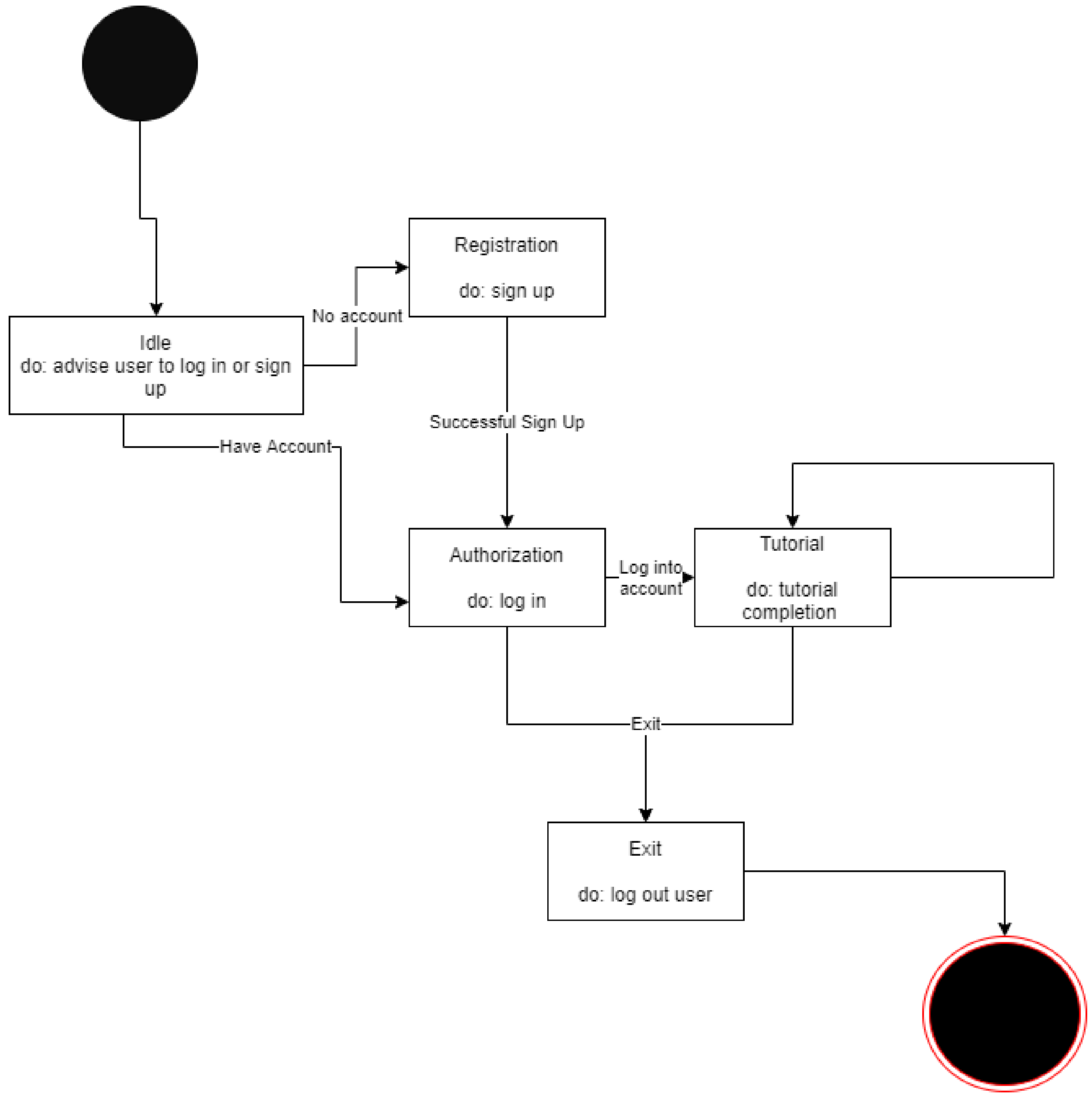
Class Models



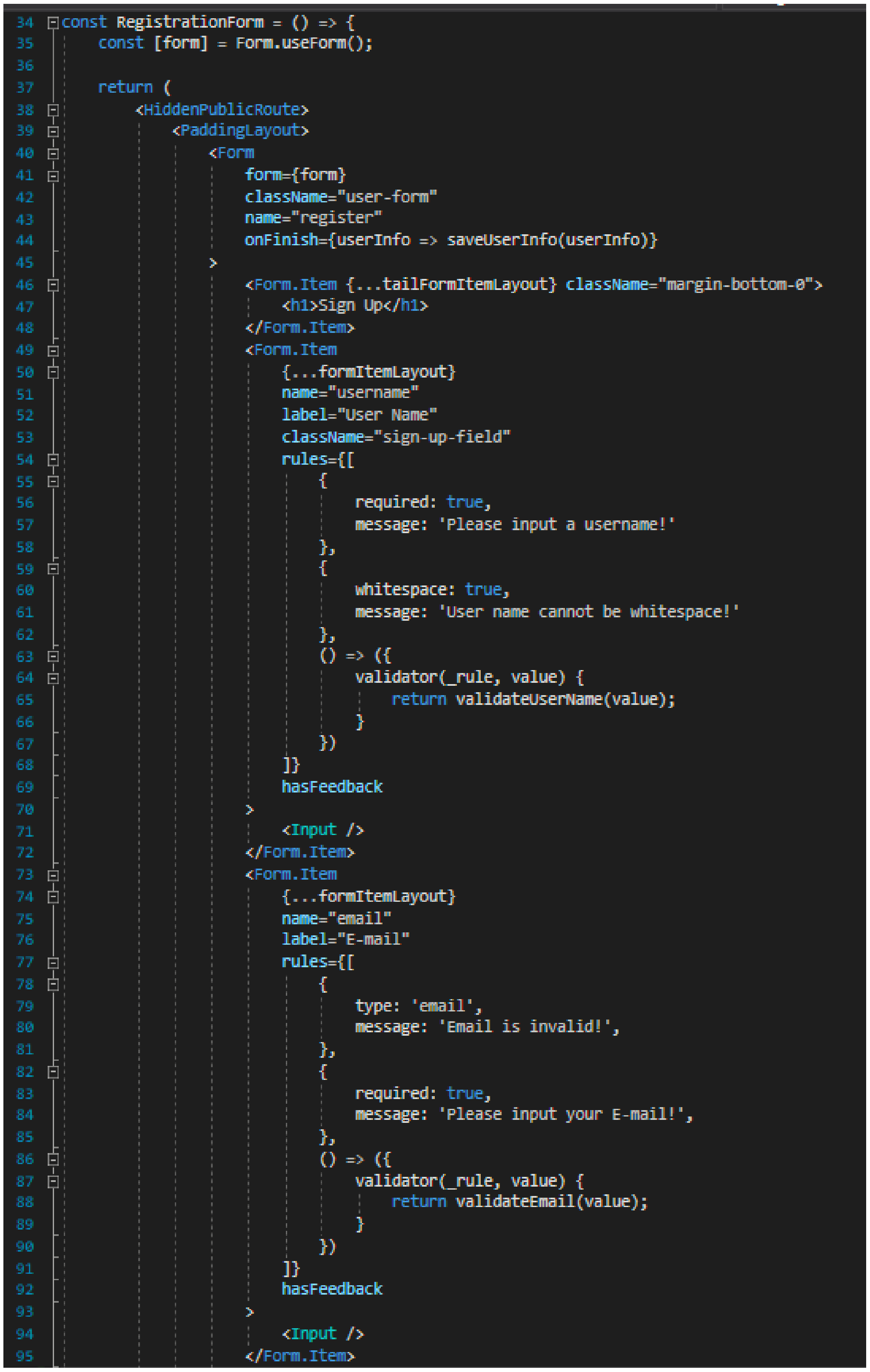
Data Models


Unauthenticated
Landing Page
Choose "Log in"
Display navigation choices
Choose "Sign up"
Sign Up
Username
Initial State
Display no error
Entered No Username
Display error: "Please input a username"
Entered Only Whitespace
Display error: "Username cannot be whitespace"
Username is Duplicate
Display error: "Username is already in use"
Valid entry
Display green checkmark
E-mail
Initial State
Display no error
Entered No E-mail
Display error: "Please input your E-mail"
Not in valid Email Format
Display error: "Email is invalid"
E-mail is Duplicate
Display error: "Email is already in use"
Valid entry
Display green checkmark
Password
Initial State
Display no error
Entered No Password
Display error: "Please input password"
Missing Upper-Case
Display error: "Password must contain an upper-case character"
Missing Lower-case
Display error: "Password must contain a lower-case character"
Missing number
Display error: "Password must contain a number"
Missing Special Character
Display error: "Password must contain one of the following special characters "?!@#$%^&*","
< 8 Characters Long
Display error: "Password must contain at least 8 characters"
Valid entry
Display green checkmark
Validate Password
Initial State
Display no error
Password Doesn't Match
Display error: "The two passwords that entered do not match"
Valid entry
Display green checkmark
Attempt registration
Confirm all fields are valid
Sign Up successful
Log In
Username or E-mail
Initial State
Display no error
Empty Input
Display error: "Please input a username!"
Non-empty
Display no error
Attempt authorization
Password
Initial State
Display no error
Empty Input
Display error: "Please input password!"
Non-empty
Display no error
Log In Attempt
Incorrect Credentials
Display error: "Those credentials are invalid"
Username/E-mail not Registered
Display error: "That username or email does not exist"
Server Error
Display error: "Sorry, there was an unexpected error. Please try again later."
All Entries Valid
Successful Log In
Navigate to Tutorial Selection Page
User chooses to sign out
Authenticated
Tutorial Selection Page
Display navigation choices
Choose "Server-Side Include Injection"
Choose "Structured Query Language Injection"
Server-Side Includes Injection Tutorial
Display ls command
Type ls command & attempt login
Display successful SSI Injection
Invalid credentials
Failed login message
Navigate to passwords.txt
Viewing paswords.txt
Type ls command & attempt login
Return to Tutorial
Knows login credentials
Valid credentials
SSI Injection Congrats! Page
Display navigation choices
Choose "Return to tutorial selection"
Choose "Go to the Structured Query Language Injection tutorial"
Structured Query Language Injection Tutorial
Display admin username
Type admin username & move to password
Display SQL query for password
Invalid credentials
Failed login message
Type correct SQL query
Display that the injection is correct
Valid credentials
Sign Out Button
Display option to "Sign Out"
SQL Injection Congrats! Page
Display navigation choices
Choose "Return to tutorial selection"

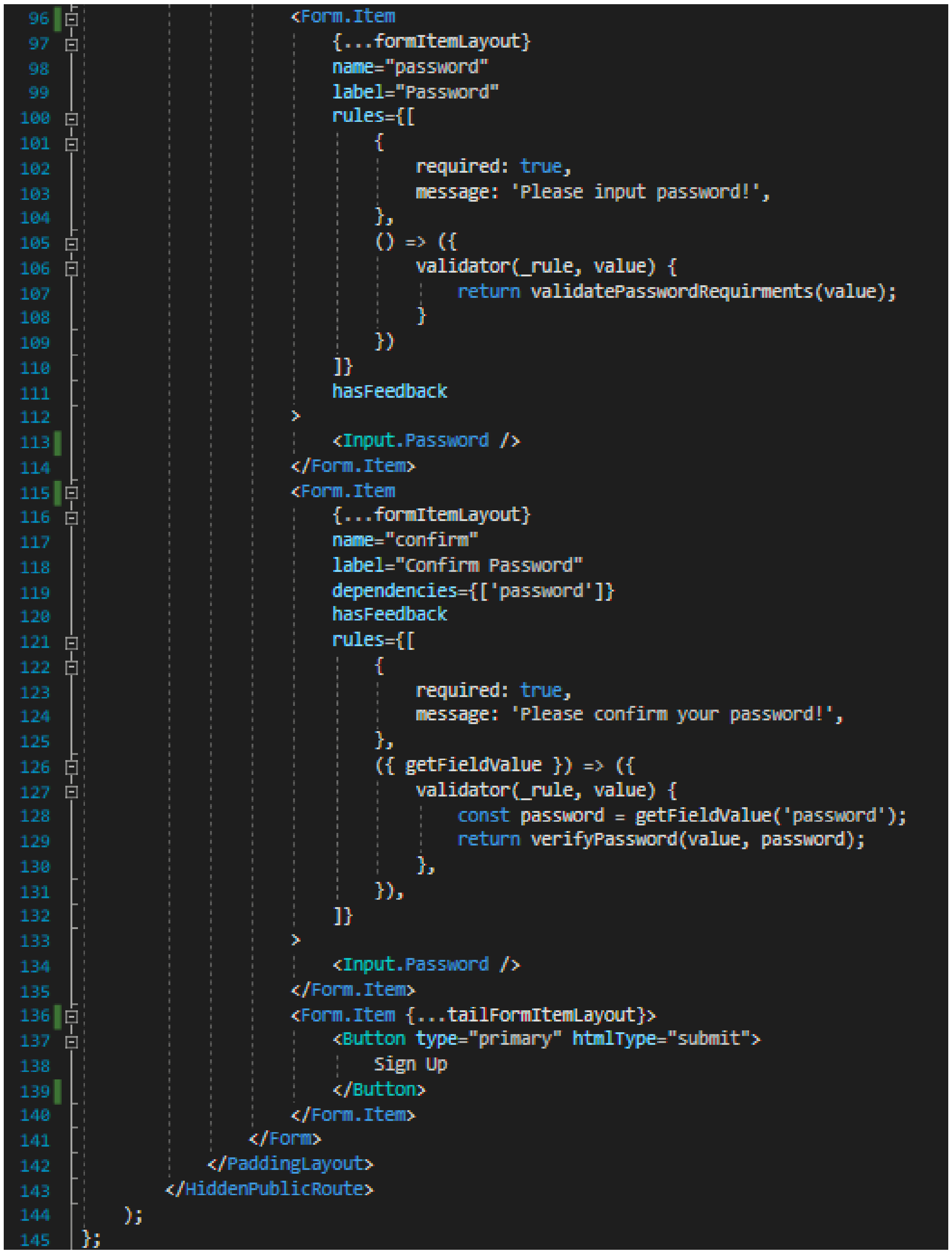
State Model


Idle
do: advise user to log in or sign up
No account
Registration
do: sign up
Successful Sign Up
Have Account
Authorization
do: log in
Log into account
Tutorial
do: tutorial completion
Exit
Exit
do: log out user

Implementation Modules

- Sign Up
    - 

```
const RegistrationForm = () => {
    const [form] = Form.useForm();

    return (
        <HiddenPublicRoute>
            <PaddingLayout>
                <Form
                    form={form}
                    className="user-form"
                    name="register"
                    onFinish={userInfo => saveUserInfo(userInfo)}
                >
                    <Form.Item {...tailFormItemLayout} className="margin-bottom-0">
                        <h1>Sign Up</h1>
                    </Form.Item>
                    <Form.Item
                        {...formItemLayout}
                        name="username"
                        label="User Name"
                        className="sign-up-field"
                        rules={[
                            {
                                required: true,
                                message: 'Please input a username!'
                            },
                            {
                                whitespace: true,
                                message: 'User name cannot be whitespace!'
                            },
                            () => ({
                                validator(_rule, value) {
                                    return validateUserName(value);
                                }
                            })
                        ]}
                        hasFeedback
                    >
                        <Input />
                    </Form.Item>
                    <Form.Item
                        {...formItemLayout}
                        name="email"
                        label="E-mail"
                        rules={[
                            {
                                type: 'email',
                                message: 'Email is invalid!',
                            },
                            {
                                required: true,
                                message: 'Please input your E-mail!',
                            },
                            () => ({
                                validator(_rule, value) {
                                    return validateEmail(value);
                                }
                            })
                        ]}
                        hasFeedback
                    >
                        <Input />
                    </Form.Item>
```



```
                <Form.Item
                    {...formItemLayout}
                    name="password"
                    label="Password"
                    rules={[
                        {
                            required: true,
                            message: 'Please input password!',
                        },
                        () => ({
                            validator(_rule, value) {
                                return validatePasswordRequirments(value);
                            }
                        })
                    ]}
                    hasFeedback
                >
                    <Input.Password />
                </Form.Item>
                <Form.Item
                    {...formItemLayout}
                    name="confirm"
                    label="Confirm Password"
                    dependencies={['password']}
                    hasFeedback
                    rules={[
                        {
                            required: true,
                            message: 'Please confirm your password!',
                        },
                        ({ getFieldValue }) => ({
                            validator(_rule, value) {
                                const password = getFieldValue('password');
                                return verifyPassword(value, password);
                            },
                        }),
                    ]}
                >
                    <Input.Password />
                </Form.Item>
                <Form.Item {...tailFormItemLayout}>
                    <Button type="primary" htmlType="submit">
                        Sign Up
                    </Button>
                </Form.Item>
            </Form>
        </PaddingLayout>
    </HiddenPublicRoute>
  );
};
```

○

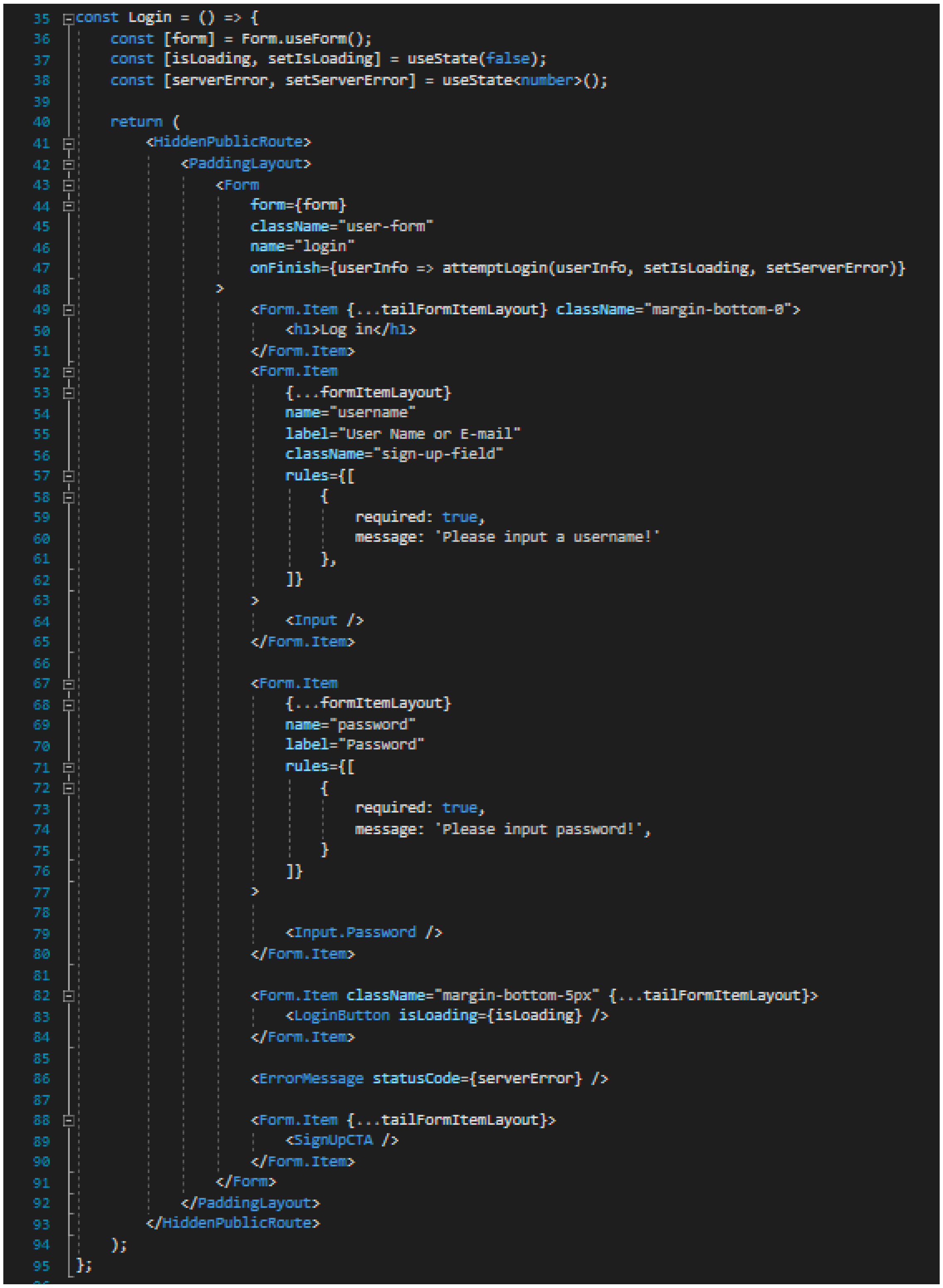
Input	Output
User provides a duplicate username.	Error message indicating the username is already in use appears below the input field. Red outline appears around the username field. Red “x” icon appears inside the username field.

	Prevents signing up.
User provides an empty or white space username.	Error message indicating the username field is empty appears below the input field. Red outline appears around the username field. Red “x” icon appears inside the username field. Prevents signing up.
User provides a duplicate email.	Error message indicating the email is already in use appears below the input field. Red outline appears around the email field. Red “x” icon appears inside the email field. Prevents signing up.
User provides an empty or white space email.	Error message indicating the email field is empty appears below the input field. Red outline appears around the email field. Red “x” icon appears inside the email field. Prevents signing up.
User provides a password without an uppercase letter.	Error message indicating the password field does not have an uppercase letter. Red outline appears around the password field. Red “x” icon appears inside the password field. Prevents signing up.
User provides a password without a lowercase letter.	Error message indicating the password field does not have a lowercase letter. Red outline appears around the password field. Red “x” icon appears inside the password field. Prevents signing up.
User provides a password without a number.	Error message indicating the password field does not have a number. Red outline appears around the password field. Red “x” icon appears inside the password

	field. Prevents signing up.
User provides a password without a symbol.	Error message indicating the password field does not have a symbol. Red outline appears around the password field. Red “x” icon appears inside the password field. Prevents signing up.
User provides a password that has less than 8 characters.	Error message indicating the password field has less than 8 characters. Red outline appears around the password field. Red “x” icon appears inside the password field. Prevents signing up.
User provides a password confirmation that doesn’t match the password field.	Error message indicating the password field does not match the password confirmation field. Red outline appears around the password confirmation field. Red “x” icon appears inside the password confirmation field. Prevents signing up.
User provides a: ● Non-empty username ● Non-duplicate username ● Non-empty email ● Non-duplicate email ● Password containing an uppercase letter ● Password containing a lowercase letter ● Password containing a number ● Password containing a symbol ● Password containing 8 or more characters ● Password confirmation that matches the password And clicks “Sign Up”	The user is signed up (their account is created) and they are brought to the login page.
User clicks the “eye” icon in the	The password field content is

password field.	shown to the user.
User clicks the “eye” icon in the password confirmation field.	The password field content is shown to the user.
User provides a valid username.	A green outline appears around the username field.
User provides a valid email.	A green outline appears around the email field.
User provides a valid password.	A green outline appears around the password field.
User provides a valid password confirmation.	A green outline appears around the password confirmation field.

- Log In



```
const Login = () => {
    const [form] = Form.useForm();
    const [isLoading, setIsLoading] = useState(false);
    const [serverError, setServerError] = useState<number>();

    return (
        <HiddenPublicRoute>
            <PaddingLayout>
                <Form
                    form={form}
                    className="user-form"
                    name="login"
                    onFinish={userInfo => attemptLogin(userInfo, setIsLoading, setServerError)}
                >
                    <Form.Item {...tailFormItemLayout} className="margin-bottom-0">
                        <h1>Log in</h1>
                    </Form.Item>
                    <Form.Item
                        {...formItemLayout}
                        name="username"
                        label="User Name or E-mail"
                        className="sign-up-field"
                        rules={[
                            {
                                required: true,
                                message: 'Please input a username!'
                            },
                        ]}
                    >
                        <Input />
                    </Form.Item>

                    <Form.Item
                        {...formItemLayout}
                        name="password"
                        label="Password"
                        rules={[
                            {
                                required: true,
                                message: 'Please input password!',
                            }
                        ]}
                    >

                        <Input.Password />
                    </Form.Item>

                    <Form.Item className="margin-bottom-5px" {...tailFormItemLayout}>
                        <LoginButton isLoading={isLoading} />
                    </Form.Item>

                    <ErrorMessage statusCode={serverError} />

                    <Form.Item {...tailFormItemLayout}>
                        <SignUpCTA />
                    </Form.Item>
                </Form>
            </PaddingLayout>
        </HiddenPublicRoute>
    );
};
```

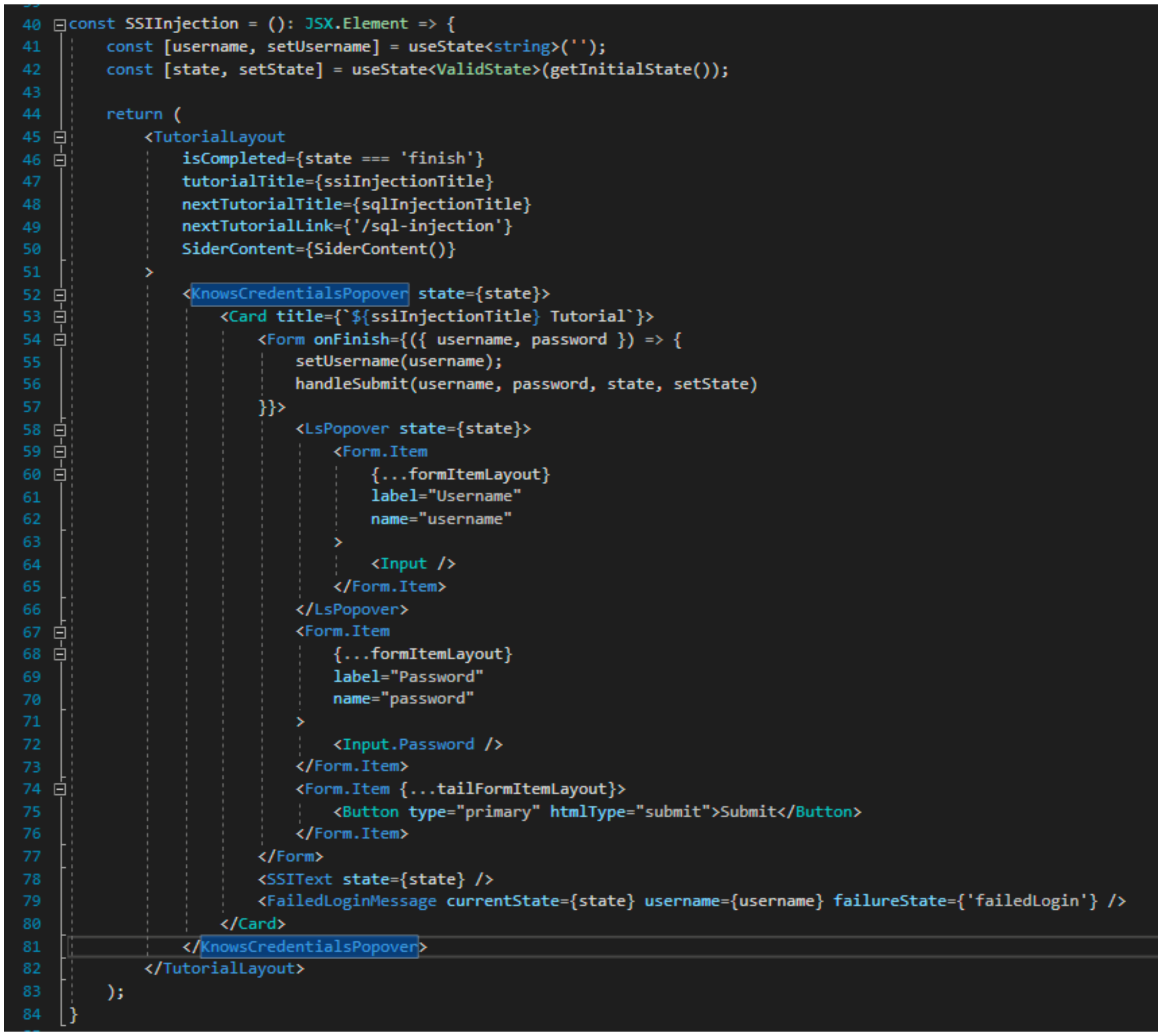
○

○

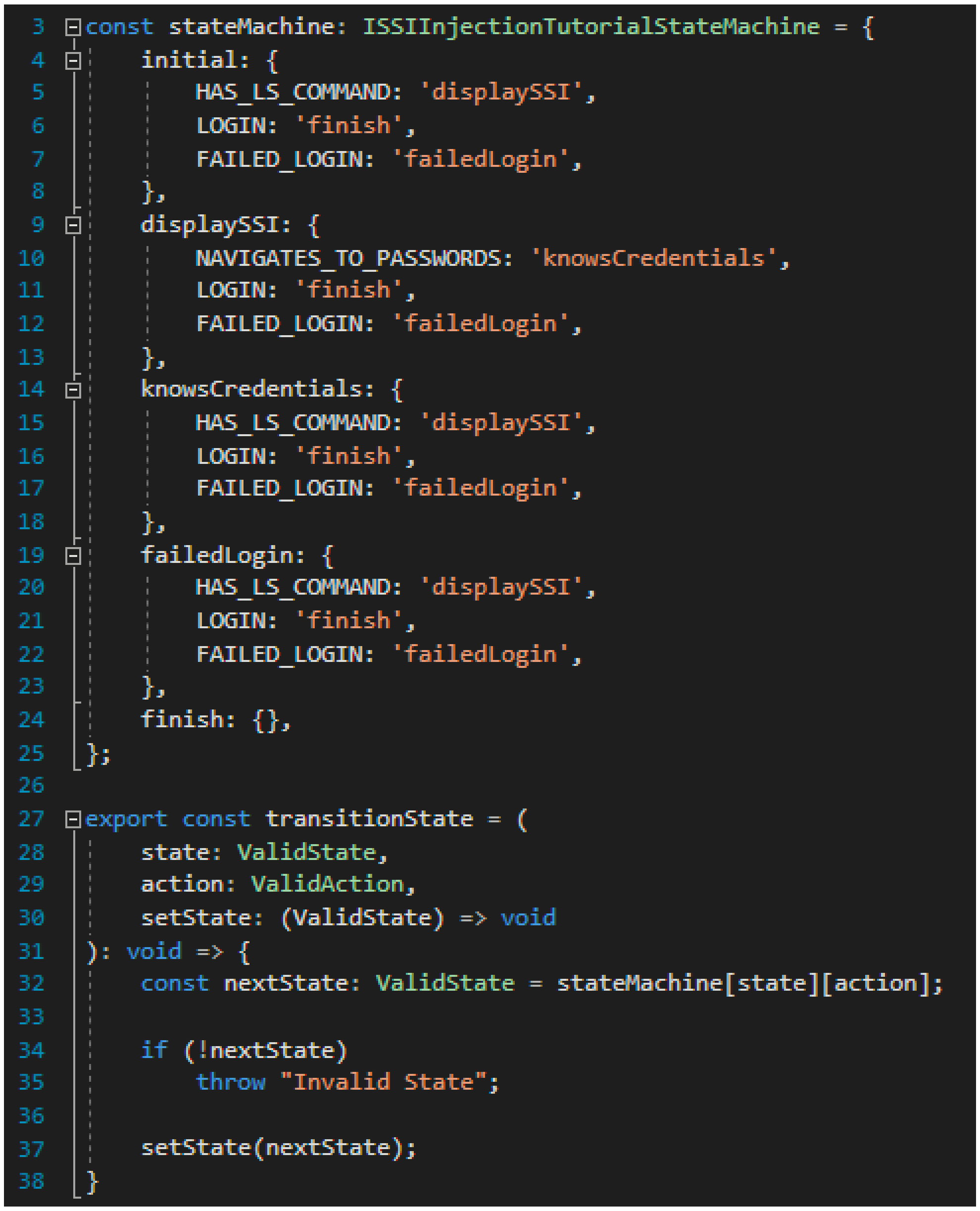
Input	Output

User enters valid username and valid password and clicks “Log In”	The user is logged into the system and brought to the tutorial selection page.
User clicks “Log In” with an empty password field.	A message appears indicating the username field is empty and the outline for the input field for password becomes red.
User enters invalid username and password combination and clicks “Log In”	A message appears indicating the username and password do not match for any user and they’re not logged in.
User clicks “Log In” with an empty username field.	A message appears indicating the username field is empty and the outline for the input field for username becomes red.
Click on the “Log In” button	Visual indication is shown that the system is processing the login attempt.
The user clicks on the “eye” icon in the password field.	The password text is shown to the user.

- SSI Injection Tutorial

◦ 

```
const SSIInjection = (): JSX.Element => {
    const [username, setUsername] = useState<string>('');
    const [state, setState] = useState<ValidState>(getInitialState());

    return (
        <TutorialLayout
            isCompleted={state === 'finish'}
            tutorialTitle={ssiInjectionTitle}
            nextTutorialTitle={sqlInjectionTitle}
            nextTutorialLink={'/sql-injection'}
            SiderContent={SiderContent()}
        >
            <KnowsCredentialsPopover state={state}>
                <Card title={`${ssiInjectionTitle} Tutorial`}>
                    <Form onFinish={({ username, password }) => {
                        setUsername(username);
                        handleSubmit(username, password, state, setState)
                    }}>
                        <LsPopover state={state}>
                            <Form.Item
                                {...formItemLayout}
                                label="Username"
                                name="username"
                            >
                                <Input />
                            </Form.Item>
                        </LsPopover>
                        <Form.Item
                            {...formItemLayout}
                            label="Password"
                            name="password"
                        >
                            <Input.Password />
                        </Form.Item>
                        <Form.Item {...tailFormItemLayout}>
                            <Button type="primary" htmlType="submit">Submit</Button>
                        </Form.Item>
                    </Form>
                    <SSIText state={state} />
                    <FailedLoginMessage currentState={state} username={username} failureState={'failedLogin'} />
                </Card>
            </KnowsCredentialsPopover>
        </TutorialLayout>
    );
}
```



```
const stateMachine: ISSIInjectionTutorialStateMachine = {
    initial: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    displaySSI: {
        NAVIGATES_TO_PASSWORDS: 'knowsCredentials',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    knowsCredentials: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    failedLogin: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    finish: {},
};

export const transitionState = (
    state: ValidState,
    action: ValidAction,
    setState: (ValidState) => void
): void => {
    const nextState: ValidState = stateMachine[state][action];

    if (!nextState)
        throw "Invalid State";

    setState(nextState);
}
```

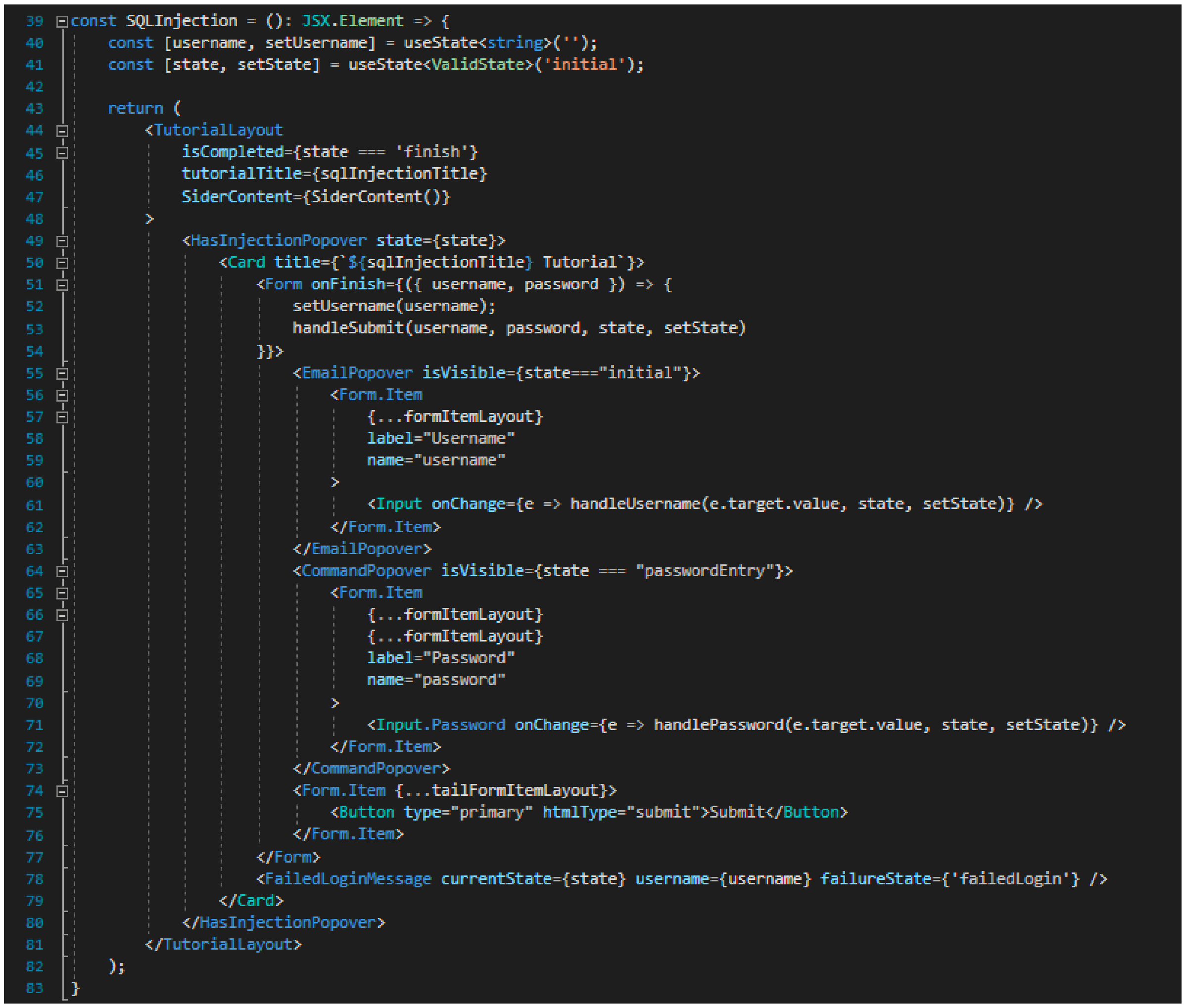
○

○

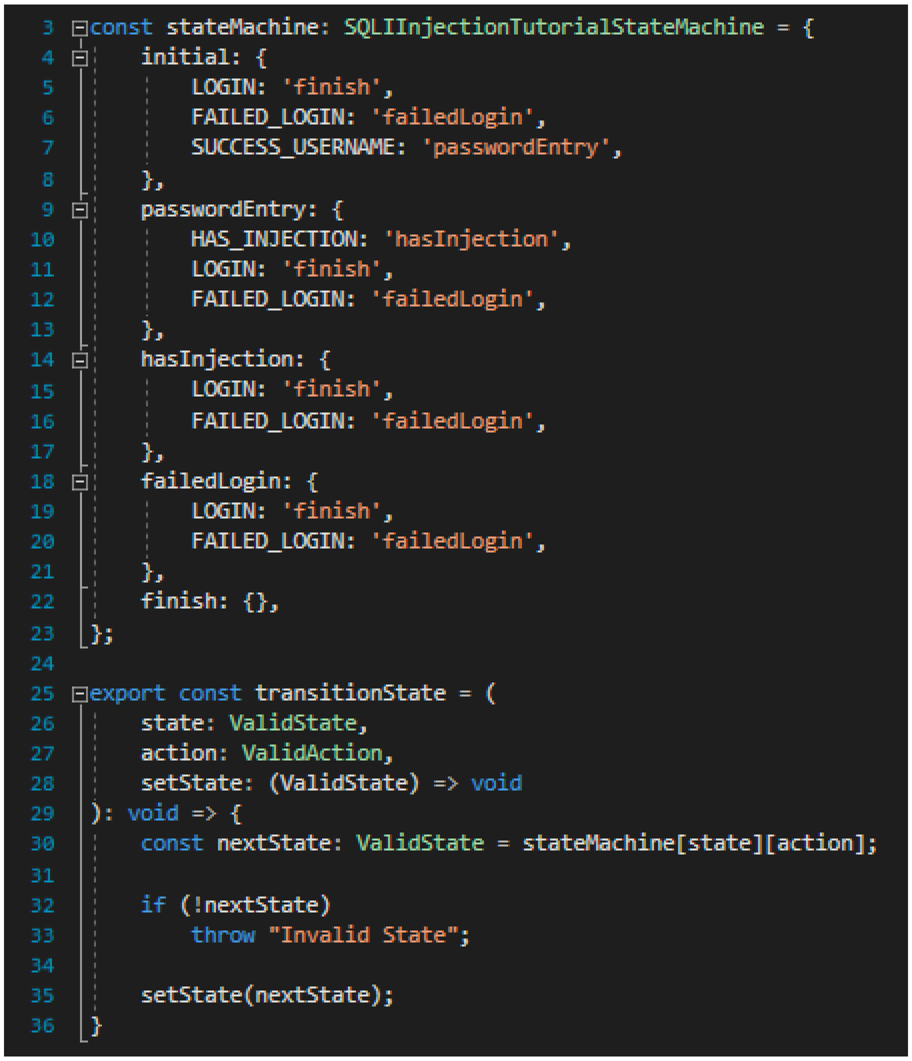
Input	Output
User navigates to the SSI Injection tutorial. “Initial” state from the “stateMachine”.	Gives the user information regarding how to perform the SSI Injection attack. Such as prompting them to enter the “ls” command in the username field and clicking “Log In”.
The user enters the “ls” command in the username and fields and	Simulated SSI Injection text is displayed to the user, indicating to

clicks “Log In”. The “displaySSI” state of the “stateMachine”.	them that they’ve done SSI Injection. Further, a new prompt is provided to them indicating that they should navigate to a file that was exposed by the SSI Injection, passwords.txt.
The user navigates to passwords.txt. The “knowsCredentials” state of the “stateMachine”.	They are given the valid username and password to be able to log in and complete the tutorial. They are also indicated that they’ve gained this information and are told how to return to the tutorial.
The user returns to the SSI Injection tutorial from passwords.txt. The “knowsCredentials” state of the “stateMachine”.	They are prompted to enter the username and password they found in passwords.txt and click “Log In”.
The user enters the correct username and password. The “finish” state of the “stateMachine”.	Tutorial is completed and the user is shown the congrats page, which gives them options to continue to the SQL Injection tutorial or go back to tutorial selection.
The user enters an invalid username or password. The “failedLogin” state of the “stateMachine”.	An error message indicating that a username with that password does not exist.

- SQL Injection Tutorial



```tsx
const SQLInjection = (): JSX.Element => {
    const [username, setUsername] = useState<string>('');
    const [state, setState] = useState<ValidState>('initial');

    return (
        <TutorialLayout
            isCompleted={state === 'finish'}
            tutorialTitle={sqlInjectionTitle}
            SiderContent={SiderContent()}
        >
            <HasInjectionPopover state={state}>
                <Card title={`${sqlInjectionTitle} Tutorial`}>
                    <Form onFinish={({ username, password }) => {
                        setUsername(username);
                        handleSubmit(username, password, state, setState)
                    }}>
                        <EmailPopover isVisible={state==="initial"}>
                            <Form.Item
                                {...formItemLayout}
                                label="Username"
                                name="username"
                            >
                                <Input onChange={e => handleUsername(e.target.value, state, setState)} />
                            </Form.Item>
                        </EmailPopover>
                        <CommandPopover isVisible={state === "passwordEntry"}>
                            <Form.Item
                                {...formItemLayout}
                                {...formItemLayout}
                                label="Password"
                                name="password"
                            >
                                <Input.Password onChange={e => handlePassword(e.target.value, state, setState)} />
                            </Form.Item>
                        </CommandPopover>
                        <Form.Item {...tailFormItemLayout}>
                            <Button type="primary" htmlType="submit">Submit</Button>
                        </Form.Item>
                    </Form>
                    <FailedLoginMessage currentState={state} username={username} failureState={'failedLogin'} />
                </Card>
            </HasInjectionPopover>
        </TutorialLayout>
    );
}
```



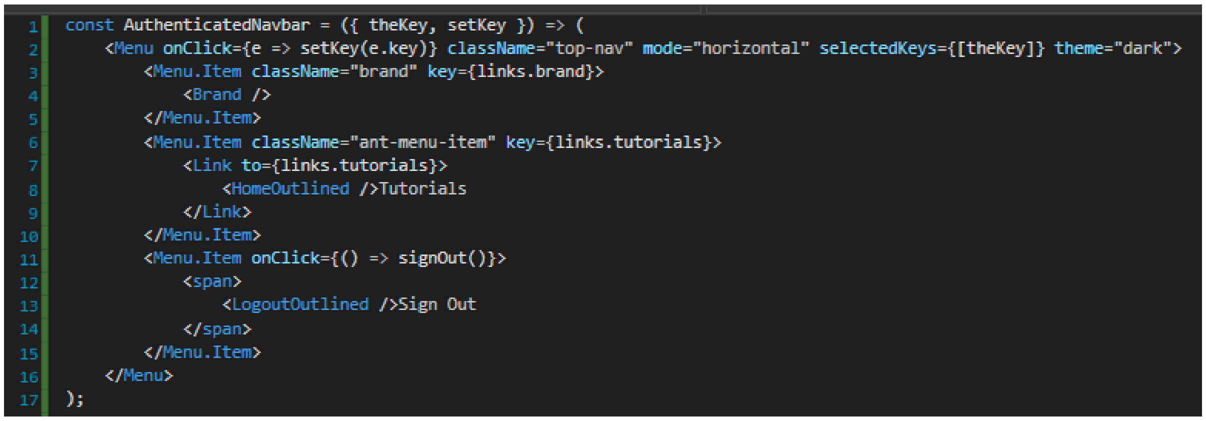
```
const stateMachine: SQLIInjectionTutorialStateMachine = {
    initial: {
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
        SUCCESS_USERNAME: 'passwordEntry',
    },
    passwordEntry: {
        HAS_INJECTION: 'hasInjection',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    hasInjection: {
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    failedLogin: {
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    finish: {},
};

export const transitionState = (
    state: ValidState,
    action: ValidAction,
    setState: (ValidState) => void
): void => {
    const nextState: ValidState = stateMachine[state][action];

    if (!nextState)
        throw "Invalid State";

    setState(nextState);
}
```

○

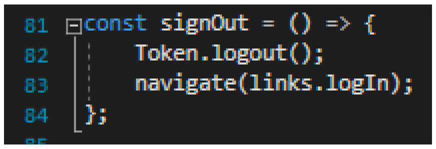
○

Input	Output
The user navigates to the SQL Injection tutorial. The “initial” state of the “stateMachine”.	Gives the user information regarding which username the attack will be targeted at.
The user enters the username that’s being targeted by the attack. The “passwordEntry” state of the “stateMachine”.	Gives the user information about how to perform the SQL Injection attack, including details about how the login is implemented under the hood and what they need to do to do the attack.
The user enters the characters that would allow for a SQL Injection	Gives the user information about the final step in completing the

attack. The “hasInjection” state of the “stateMachine.	attack, clicking “Log In”.
The user clicks the “Log In” button with the correct username and password (the injection). The “finish” state of the “stateMachine”.	Tutorial is completed and the user is shown the congrats page, which gives them an option to return to the tutorial selection page.
The user enters an invalid username or password. The “failedLogin” state of the “stateMachine”.	An error message indicating that a username with that password does not exist.

- Sign Out
    - 

```
const AuthenticatedNavbar = ({ theKey, setKey }) => (
    <Menu onClick={e => setKey(e.key)} className="top-nav" mode="horizontal" selectedKeys={[theKey]} theme="dark">
        <Menu.Item className="brand" key={links.brand}>
            <Brand />
        </Menu.Item>
        <Menu.Item className="ant-menu-item" key={links.tutorials}>
            <Link to={links.tutorials}>
                <HomeOutlined />Tutorials
            </Link>
        </Menu.Item>
        <Menu.Item onClick={() => signOut()}>
            <span>
                <LogoutOutlined />Sign Out
            </span>
        </Menu.Item>
    </Menu>
);
```

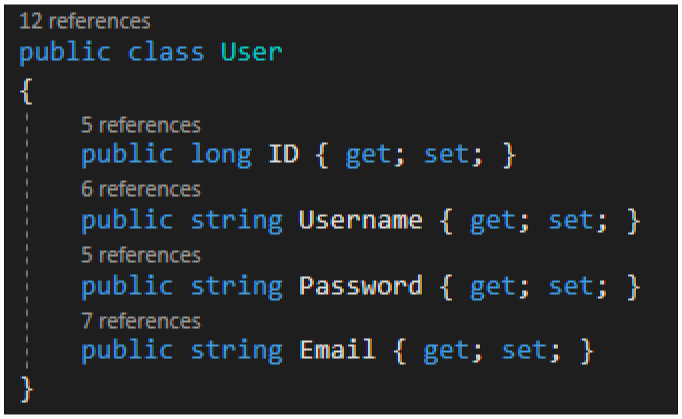
    - 

```
const signOut = () => {
    Token.logout();
    navigate(links.logIn);
};
```

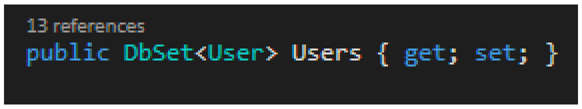
    -

Input	Output
User logs in to the system.	“Sign Out” appears as an option on the navbar.
User clicks “Sign Out” on the navbar.	The user is logged out of the system (their token is removed locally) and they’re redirected to the login page. Visual indication is also given for clicking “Sign Out”.

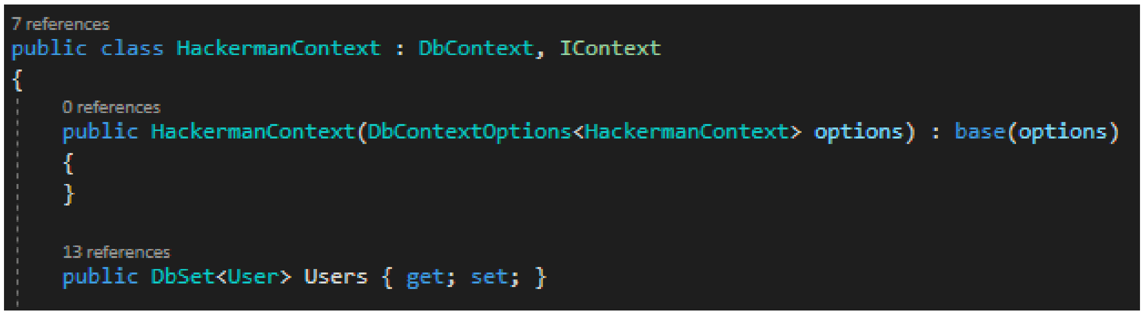
Design Patterns

- Repository
    - Type: structural
    - 

```
12 references
public class User
{
    5 references
    public long ID { get; set; }
    6 references
    public string Username { get; set; }
    5 references
    public string Password { get; set; }
    7 references
    public string Email { get; set; }
}
```

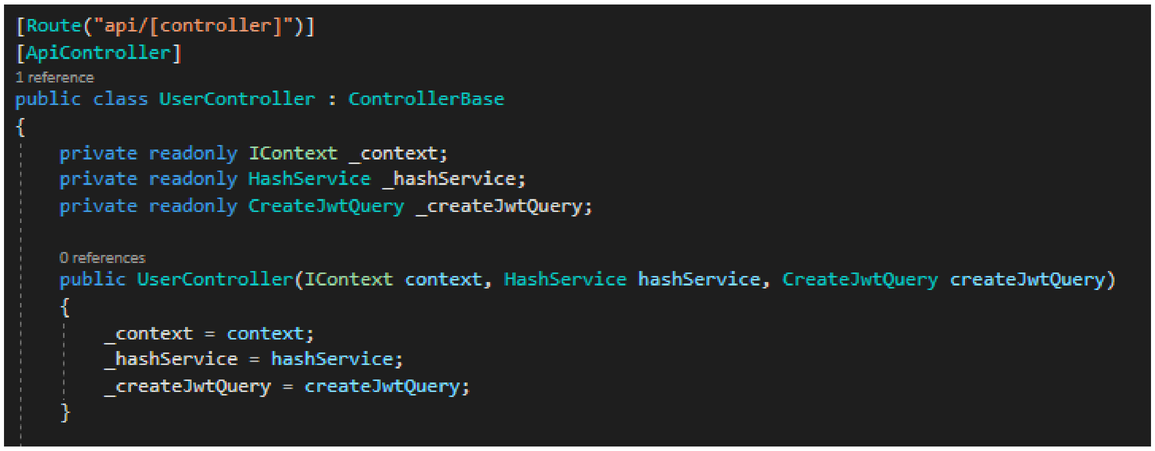
    - 

```
13 references
public DbSet<User> Users { get; set; }
```

    - Description: In the above the DbSet for User is a repository. The repository pattern is used here by our ORM, EntityFramework Core, to abstract away the low-level database queries.
- Unit of Work
    - Type: behavioral
    - 

```
7 references
public class HackermanContext : DbContext, IContext
{
    0 references
    public HackermanContext(DbContextOptions<HackermanContext> options) : base(options)
    {
    }

    13 references
    public DbSet<User> Users { get; set; }
```

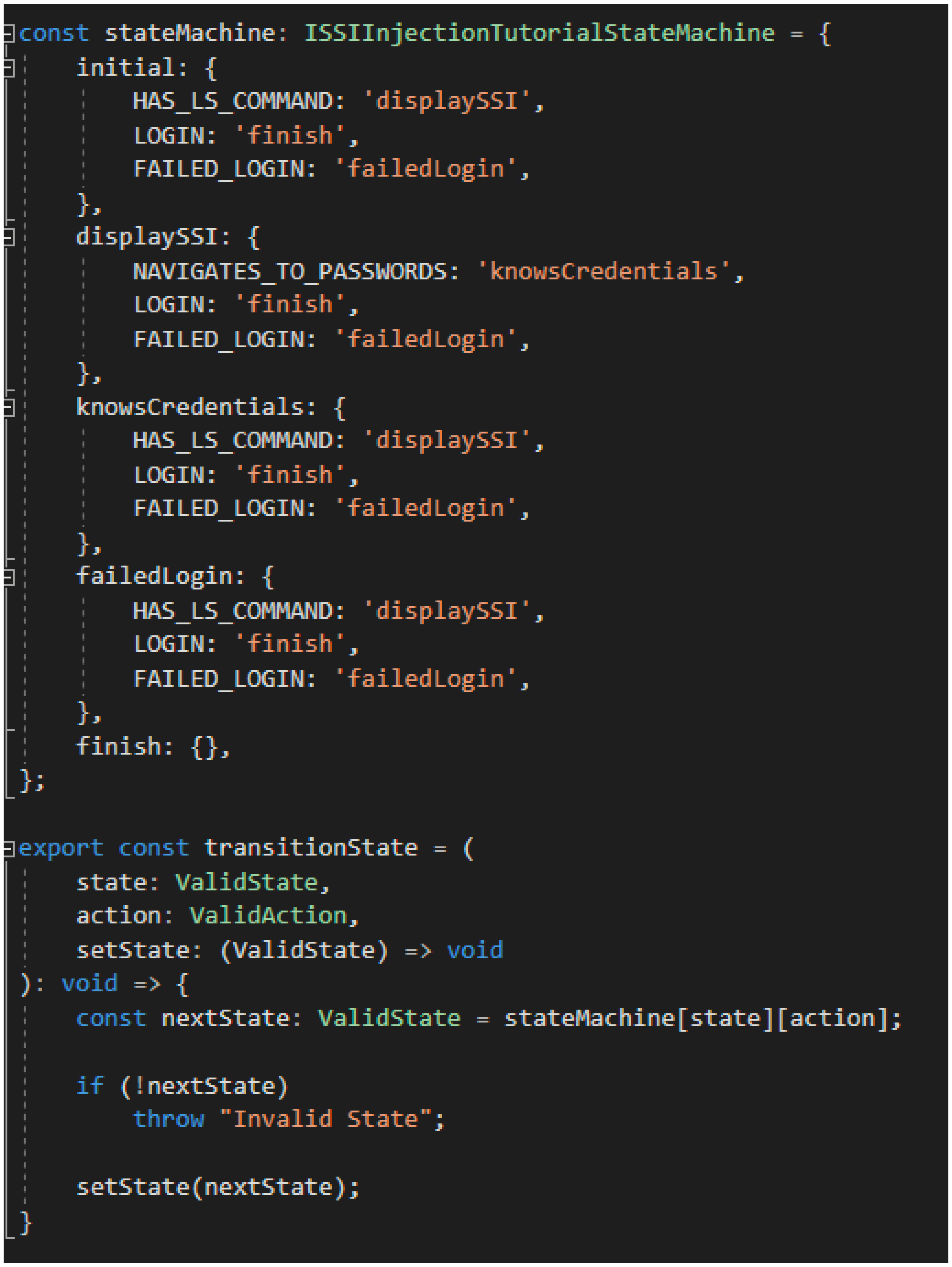
    - Description: Our DbContext sub-class, HackermanContext uses Unit of Work to atomically perform database insert/update/delete operations. This is of course important for performance reasons, but also for simplicity's sake. This is handled under the hood by our ORM, EntityFramework Core.
- Dependency Injection
    - Type: creational

- 

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class UserController : ControllerBase
{
    private readonly IContext _context;
    private readonly HashService _hashService;
    private readonly CreateJwtQuery _createJwtQuery;

    0 references
    public UserController(IContext context, HashService hashService, CreateJwtQuery createJwtQuery)
    {
        _context = context;
        _hashService = hashService;
        _createJwtQuery = createJwtQuery;
    }
```

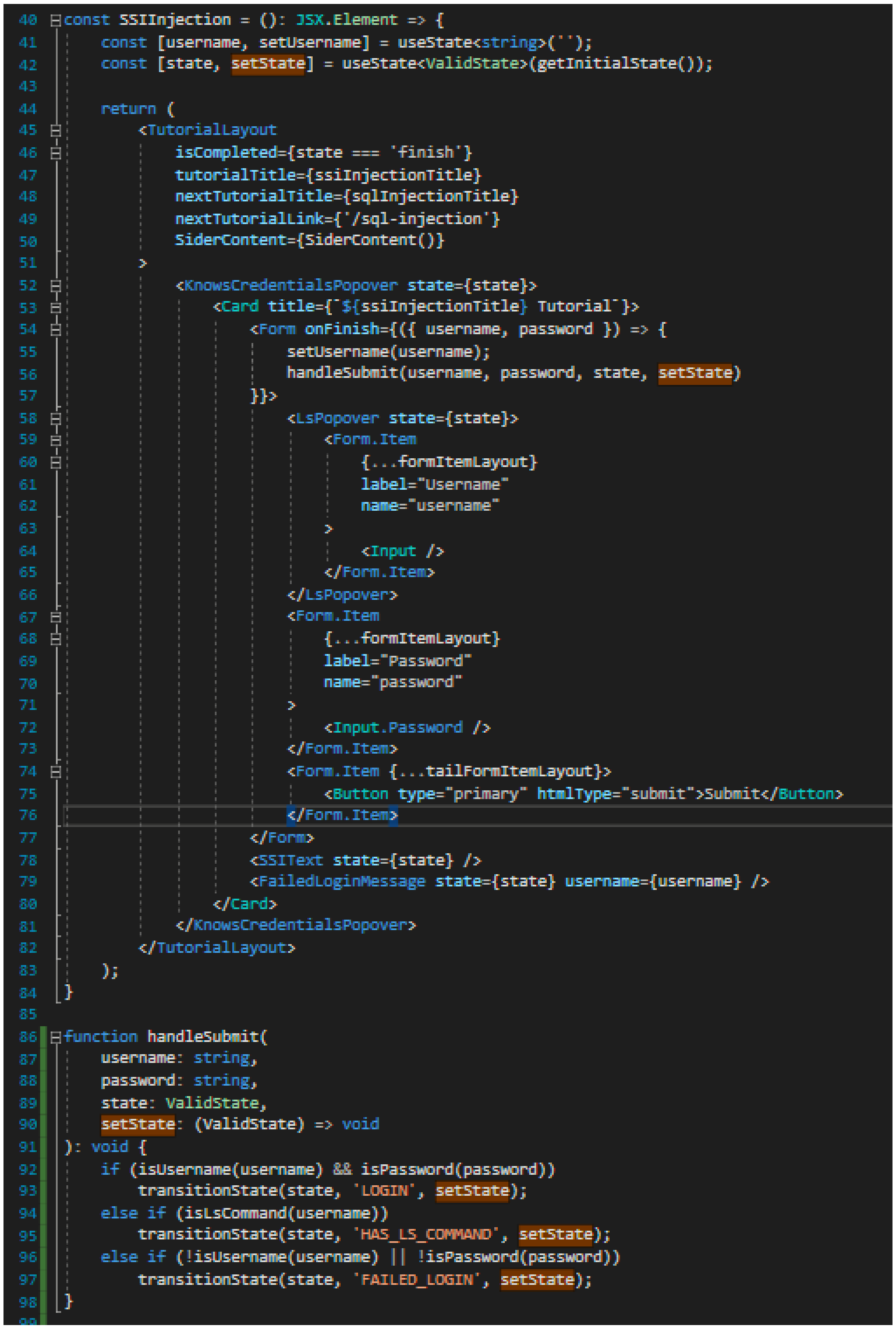
- Description: We use dependency injection to make our code more readable, flexible, and testable. This is done by injecting something via an argument to a function of some sort. For example, in the above we can now more easily mock “IContext” (which is used for database operations) for testing and this is important because we don’t want our unit tests to hit the database, we instead want them to use data that we created for the purpose of testing.

- Finite State Machine (State)
  - Type: behavioral



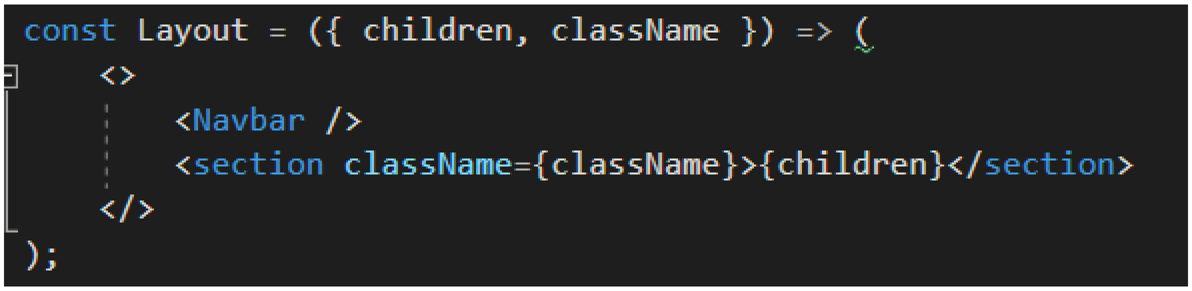
```
const stateMachine: ISSIInjectionTutorialStateMachine = {
    initial: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    displaySSI: {
        NAVIGATES_TO_PASSWORDS: 'knowsCredentials',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    knowsCredentials: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    failedLogin: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    finish: {},
};

export const transitionState = (
    state: ValidState,
    action: ValidAction,
    setState: (ValidState) => void
): void => {
    const nextState: ValidState = stateMachine[state][action];

    if (!nextState)
        throw "Invalid State";

    setState(nextState);
}
```

○

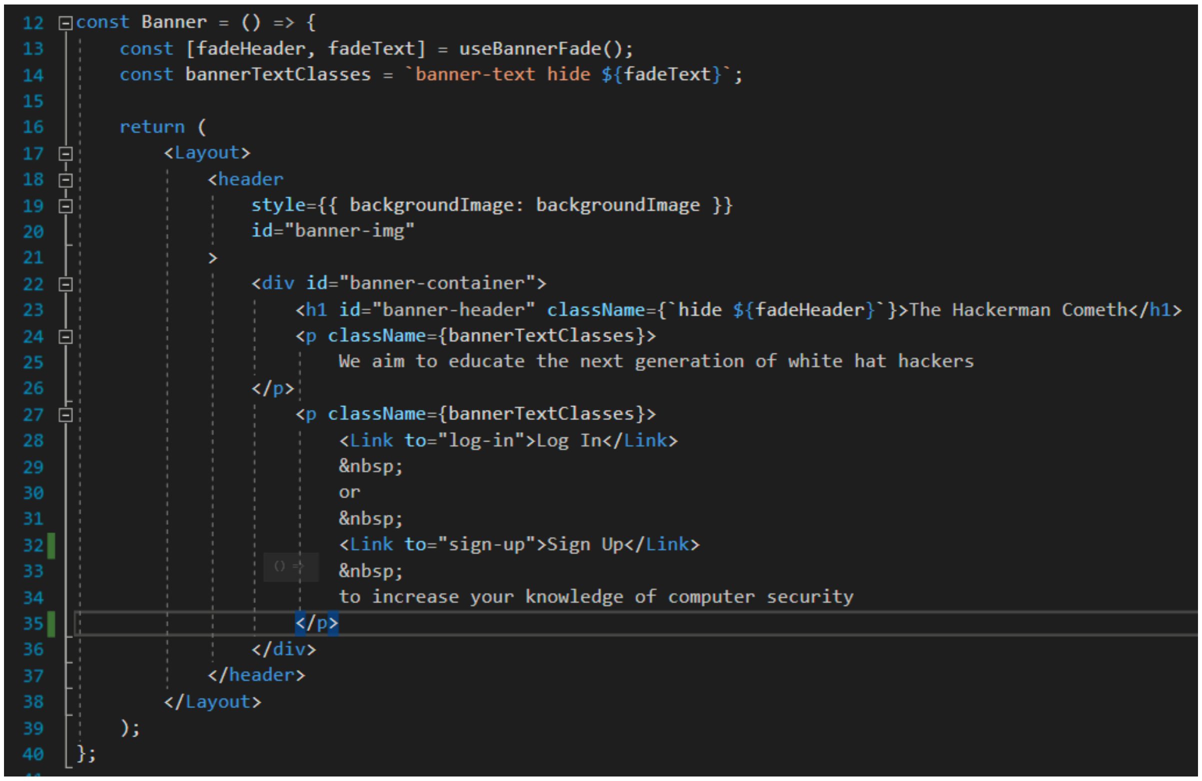


```
const SSIInjection = (): JSX.Element => {
    const [username, setUsername] = useState<string>('');
    const [state, setState] = useState<ValidState>(getInitialState());

    return (
        <TutorialLayout
            isCompleted={state === 'finish'}
            tutorialTitle={ssiInjectionTitle}
            nextTutorialTitle={sqlInjectionTitle}
            nextTutorialLink={'/sql-injection'}
            SiderContent={SiderContent()}
        >
            <KnowsCredentialsPopover state={state}>
                <Card title={`${ssiInjectionTitle} Tutorial`}>
                    <Form onFinish={({ username, password }) => {
                        setUsername(username);
                        handleSubmit(username, password, state, setState)
                    }}>
                        <LsPopover state={state}>
                            <Form.Item
                                {...formItemLayout}
                                label="Username"
                                name="username"
                            >
                                <Input />
                            </Form.Item>
                        </LsPopover>
                        <Form.Item
                            {...formItemLayout}
                            label="Password"
                            name="password"
                        >
                            <Input.Password />
                        </Form.Item>
                        <Form.Item {...tailFormItemLayout}>
                            <Button type="primary" htmlType="submit">Submit</Button>
                        </Form.Item>
                    </Form>
                    <SSIText state={state} />
                    <FailedLoginMessage state={state} username={username} />
                </Card>
            </KnowsCredentialsPopover>
        </TutorialLayout>
    );
}

function handleSubmit(
    username: string,
    password: string,
    state: ValidState,
    setState: (ValidState) => void
): void {
    if (isUsername(username) && isPassword(password))
        transitionState(state, 'LOGIN', setState);
    else if (isLsCommand(username))
        transitionState(state, 'HAS_LS_COMMAND', setState);
    else if (!isUsername(username) || !isPassword(password))
        transitionState(state, 'FAILED_LOGIN', setState);
}
```

- Description: A finite state machine allows us to explicitly state our states and transitions between those states. This pattern is used to prevent bugs and provide more readable and scalable code. For example, in the above we have a state machine, in that state machine there's a state called “initial”, that “initial” state can only transition to the “displaySSI”, “finish”, or “failedLogin” states. The UI is rendered based on these explicit states.

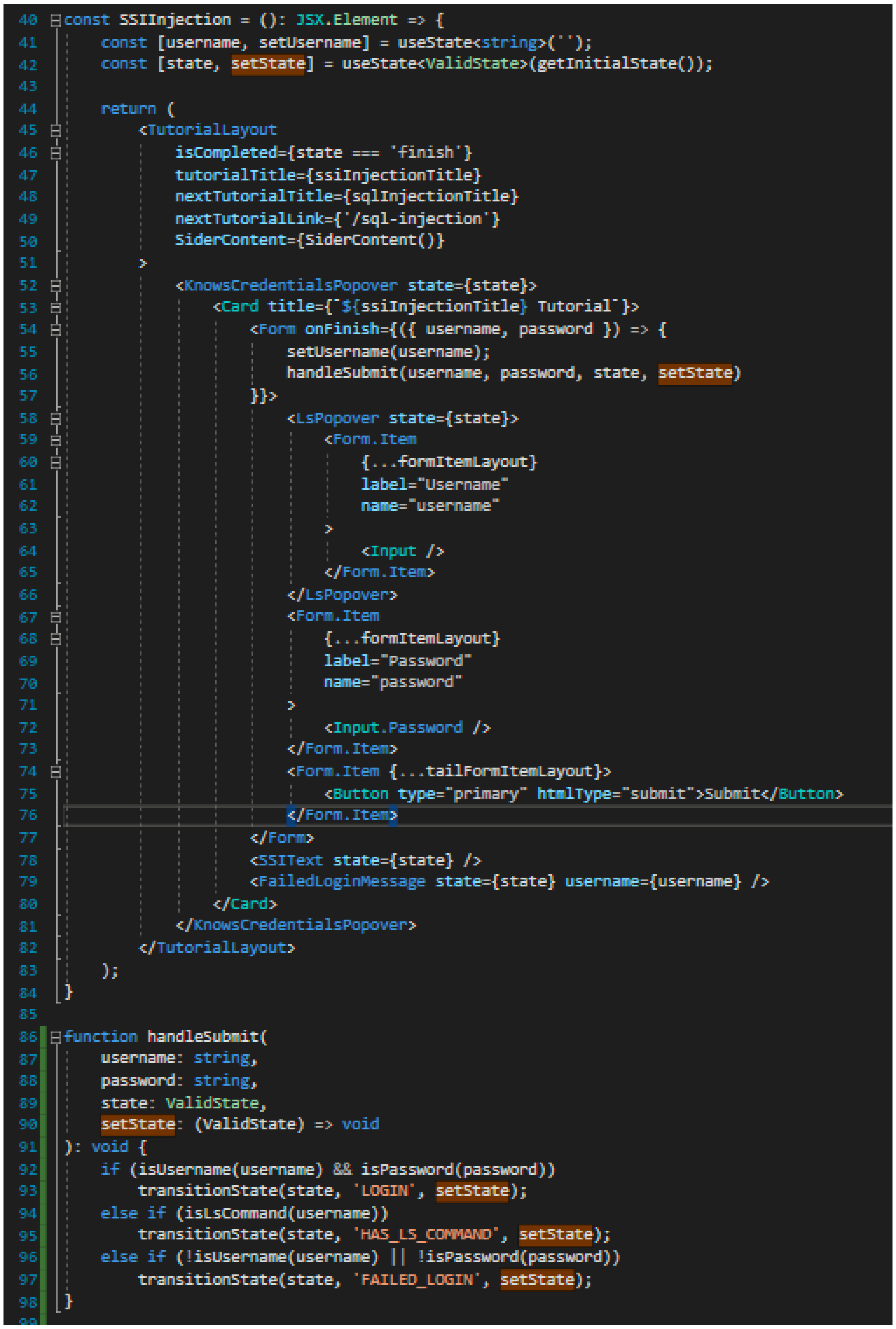
- Layout Component
    - Type: structural
    - 

```
const Layout = ({ children, className }) => (
    <>
        <Navbar />
        <section className={className}>{children}</section>
    </>
);
```

    - 

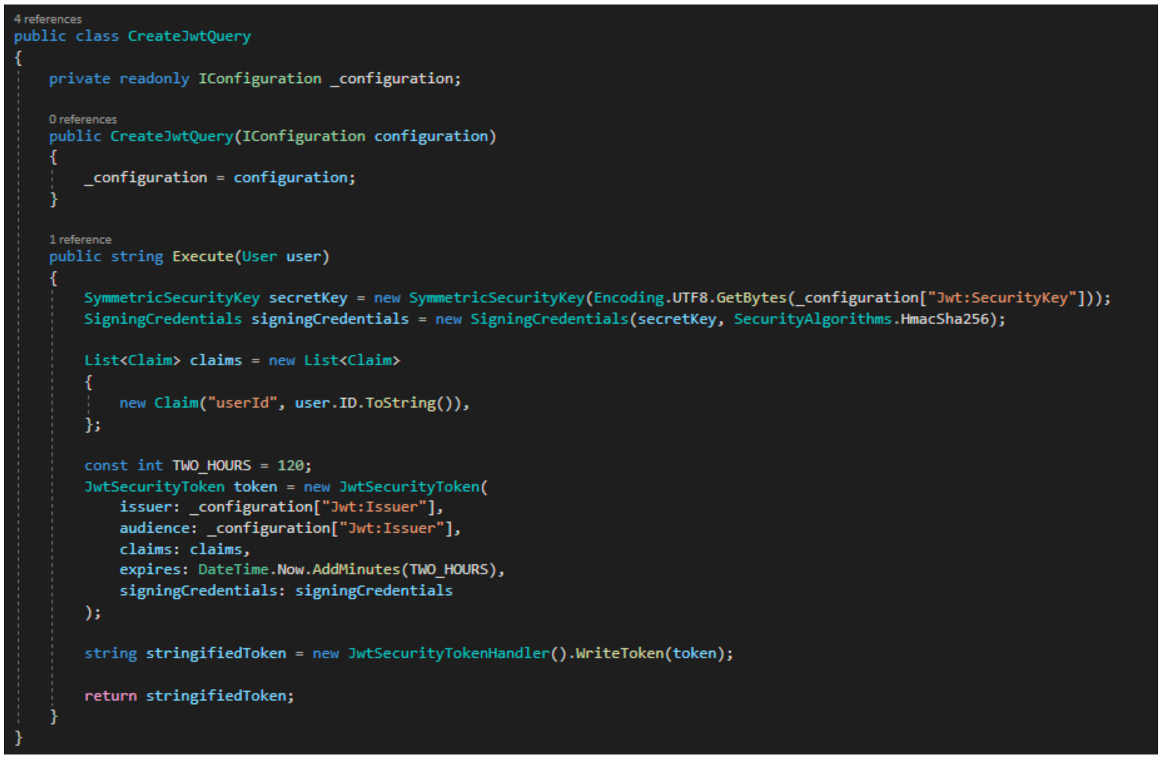
```
const Banner = () => {
    const [fadeHeader, fadeText] = useBannerFade();
    const bannerTextClasses = `banner-text hide ${fadeText}`;

    return (
        <Layout>
            <header
                style={{ backgroundImage: backgroundImage }}
                id="banner-img"
            >
                <div id="banner-container">
                    <h1 id="banner-header" className={`hide ${fadeHeader}`}>The Hackerman Cometh</h1>
                    <p className={bannerTextClasses}>
                        We aim to educate the next generation of white hat hackers
                </p>
                    <p className={bannerTextClasses}>
                        <Link to="log-in">Log In</Link>
                        &nbsp;
                        or
                        &nbsp;
                        <Link to="sign-up">Sign Up</Link>
                        &nbsp;
                        to increase your knowledge of computer security
                    </p>
                </div>
            </header>
        </Layout>
    );
};
```

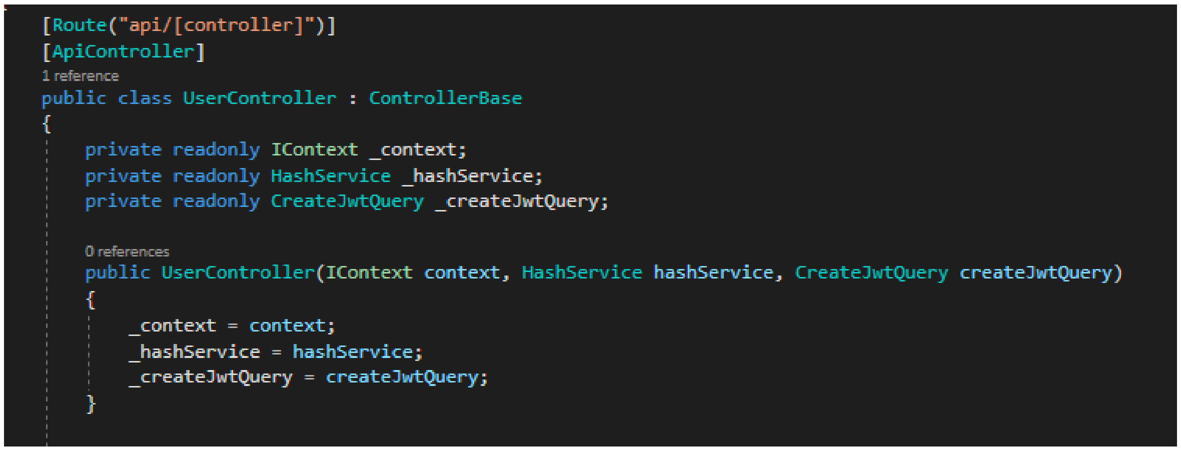
    - Description: A Layout Component allows us to specify common markup and logic to our site without having to duplicate it individually across all of our components. In this case our layout currently only has a navbar and the normal content, but in the future if we wanted to add something else like a footer we could do so and only have to do it in 1 place. In the above we wrap the Banner with our Layout Component so that the page has a navbar and in the future could support other UI elements without changing the code in Banner. We use this pattern frequently across our codebase, it’s also how we limit duplication for our tutorials (sidebar content, faded overlay, practice button, congrats page, and general formatting).

- State Hoisting
    - Type: behavioral
    - 

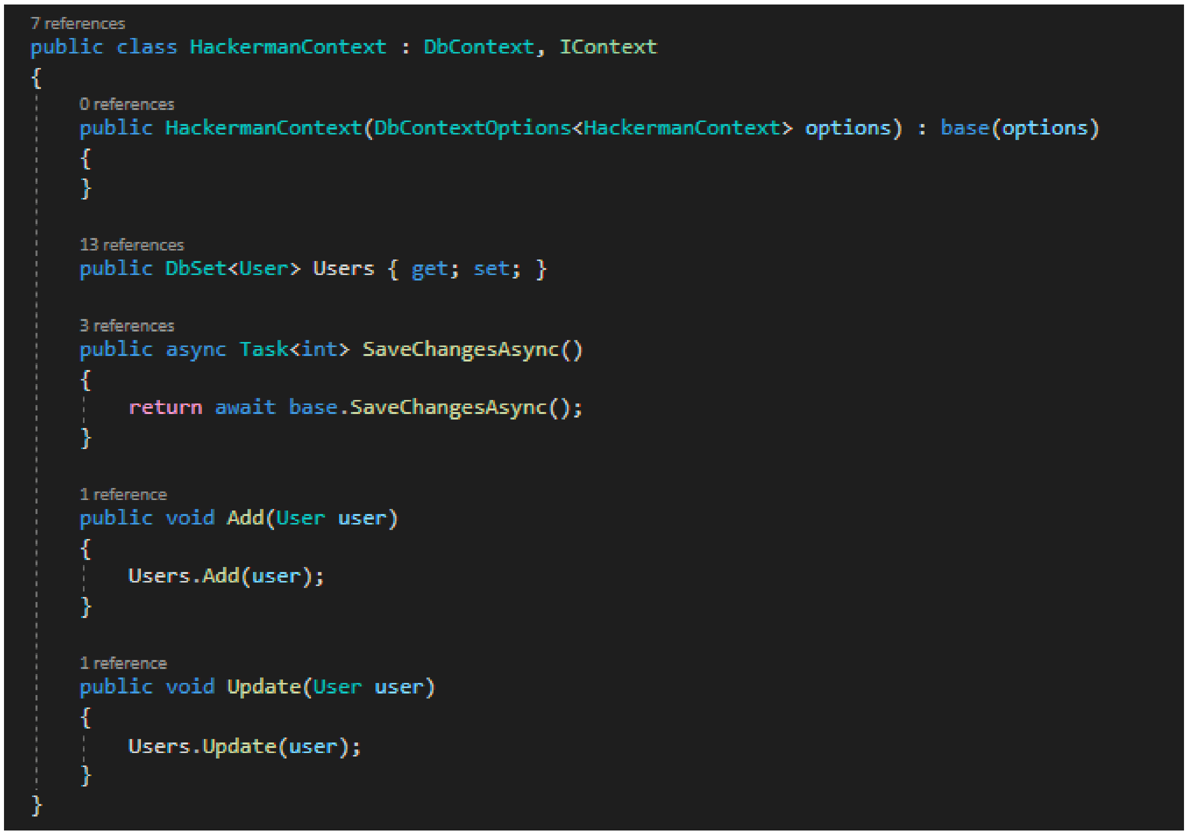
```
const SSIInjection = (): JSX.Element => {
    const [username, setUsername] = useState<string>('');
    const [state, setState] = useState<ValidState>(getInitialState());

    return (
        <TutorialLayout
            isCompleted={state === 'finish'}
            tutorialTitle={ssiInjectionTitle}
            nextTutorialTitle={sqlInjectionTitle}
            nextTutorialLink={'/sql-injection'}
            SiderContent={SiderContent()}
        >
            <KnowsCredentialsPopover state={state}>
                <Card title={`${ssiInjectionTitle} Tutorial`}>
                    <Form onFinish={({ username, password }) => {
                        setUsername(username);
                        handleSubmit(username, password, state, setState)
                    }}>
                        <LsPopover state={state}>
                            <Form.Item
                                {...formItemLayout}
                                label="Username"
                                name="username"
                            >
                                <Input />
                            </Form.Item>
                        </LsPopover>
                        <Form.Item
                            {...formItemLayout}
                            label="Password"
                            name="password"
                        >
                            <Input.Password />
                        </Form.Item>
                        <Form.Item {...tailFormItemLayout}>
                            <Button type="primary" htmlType="submit">Submit</Button>
                        </Form.Item>
                    </Form>
                    <SSIText state={state} />
                    <FailedLoginMessage state={state} username={username} />
                </Card>
            </KnowsCredentialsPopover>
        </TutorialLayout>
    );
}

function handleSubmit(
    username: string,
    password: string,
    state: ValidState,
    setState: (ValidState) => void
): void {
    if (isUsername(username) && isPassword(password))
        transitionState(state, 'LOGIN', setState);
    else if (isLsCommand(username))
        transitionState(state, 'HAS_LS_COMMAND', setState);
    else if (!isUsername(username) || !isPassword(password))
        transitionState(state, 'FAILED_LOGIN', setState);
}
```

    - Description: This pattern allows sub-components to update the state of sibling components. This is accomplished by moving that state outside of the sub-components and into the shared parent component. The parent can then pass down the state change operations as props to the sub-components, allowing siblings, in essence, to update the state of 1 another. In the above example, the “state” and “setState” variables were hoisted out of the sub-components (Input) into the parent component (SSIInjection).

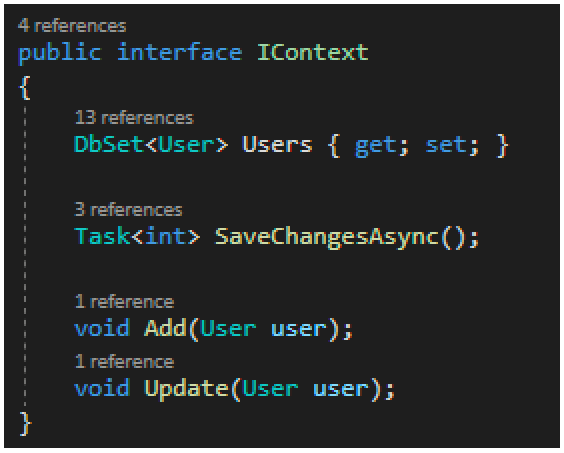
- Command/Query
    - Type: structural
      
      ```
      4 references
      public class CreateJwtQuery
      {
          private readonly IConfiguration _configuration;

          0 references
          public CreateJwtQuery(IConfiguration configuration)
          {
              _configuration = configuration;
          }

          1 reference
          public string Execute(User user)
          {
              SymmetricSecurityKey secretKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:SecurityKey"]));
              SigningCredentials signingCredentials = new SigningCredentials(secretKey, SecurityAlgorithms.HmacSha256);

              List<Claim> claims = new List<Claim>
              {
                  new Claim("userId", user.ID.ToString()),
              };

              const int TWO_HOURS = 120;
              JwtSecurityToken token = new JwtSecurityToken(
                  issuer: _configuration["Jwt:Issuer"],
                  audience: _configuration["Jwt:Issuer"],
                  claims: claims,
                  expires: DateTime.Now.AddMinutes(TWO_HOURS),
                  signingCredentials: signingCredentials
              );

              string stringifiedToken = new JwtSecurityTokenHandler().WriteToken(token);

              return stringifiedToken;
          }
      }
      ```
    - Description: The “CreateJwtQuery” creates a json web token following the command/query pattern. The pattern in this case is useful to constrain how much can happen in the class, it puts emphasis on the single responsibility principle.
- Dependency Inversion - Interface Indirection
    - Type: structural

- 

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class UserController : ControllerBase
{
    private readonly IContext _context;
    private readonly HashService _hashService;
    private readonly CreateJwtQuery _createJwtQuery;

    0 references
    public UserController(IContext context, HashService hashService, CreateJwtQuery createJwtQuery)
    {
        _context = context;
        _hashService = hashService;
        _createJwtQuery = createJwtQuery;
    }
```

- 

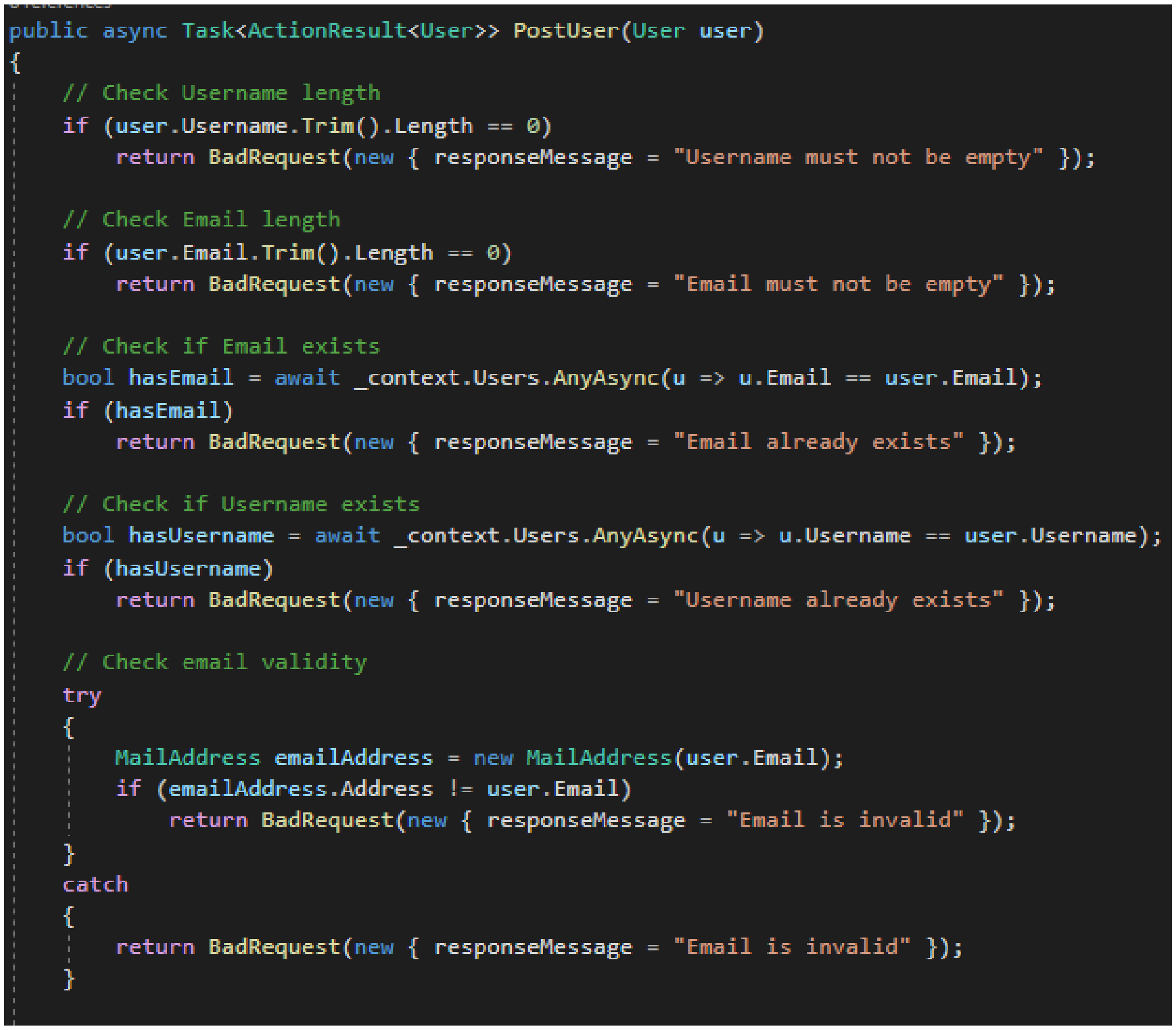
```
7 references
public class HackermanContext : DbContext, IContext
{
    0 references
    public HackermanContext(DbContextOptions<HackermanContext> options) : base(options)
    {
    }

    13 references
    public DbSet<User> Users { get; set; }

    3 references
    public async Task<int> SaveChangesAsync()
    {
        return await base.SaveChangesAsync();
    }

    1 reference
    public void Add(User user)
    {
        Users.Add(user);
    }

    1 reference
    public void Update(User user)
    {
        Users.Update(user);
    }
}
```

- 

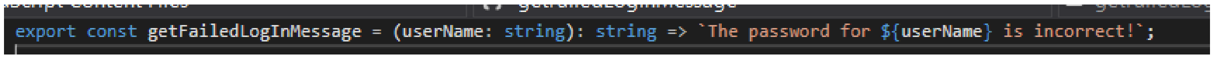
```
4 references
public interface IContext
{
    13 references
    DbSet<User> Users { get; set; }

    3 references
    Task<int> SaveChangesAsync();

    1 reference
    void Add(User user);
    1 reference
    void Update(User user);
}
```

- Description: “HackermanContext” uses DbContext and implements “IContext”. This allows for dependency inversion (i.e. high-level modules should not depend on low-level modules), which most importantly means that we could easily switch out our “IContext” implementation for some sort of other abstraction that interacts with the database if that need ever arises. This also makes unit testing much easier, especially since we’re also using dependency injection.

Cohesion

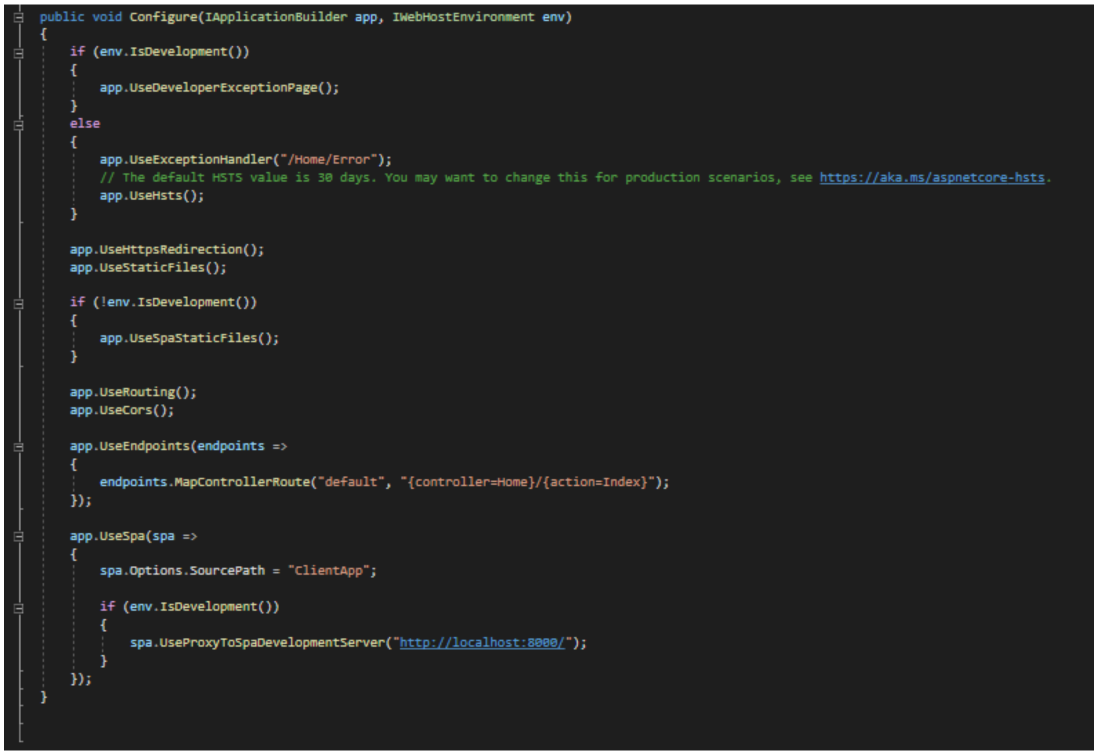
- Type: procedural
    - 

```
public async Task<ActionResult<User>> PostUser(User user)
{
    // Check Username length
    if (user.Username.Trim().Length == 0)
        return BadRequest(new { responseMessage = "Username must not be empty" });

    // Check Email length
    if (user.Email.Trim().Length == 0)
        return BadRequest(new { responseMessage = "Email must not be empty" });

    // Check if Email exists
    bool hasEmail = await _context.Users.AnyAsync(u => u.Email == user.Email);
    if (hasEmail)
        return BadRequest(new { responseMessage = "Email already exists" });

    // Check if Username exists
    bool hasUsername = await _context.Users.AnyAsync(u => u.Username == user.Username);
    if (hasUsername)
        return BadRequest(new { responseMessage = "Username already exists" });

    // Check email validity
    try
    {
        MailAddress emailAddress = new MailAddress(user.Email);
        if (emailAddress.Address != user.Email)
            return BadRequest(new { responseMessage = "Email is invalid" });
    }
    catch
    {
        return BadRequest(new { responseMessage = "Email is invalid" });
    }
```

    - Description: This shows procedural cohesion because it follows a sequence of events for validating a user's credentials before successfully signing them up.
- Type: functional
    - 

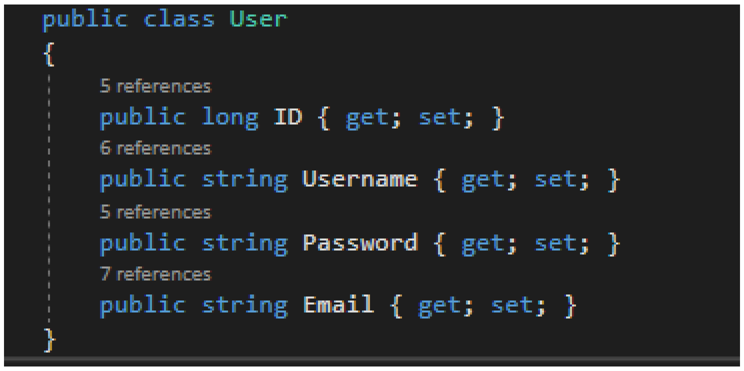
```
export const getFailedLogInMessage = (userName: string): string => `The password for ${userName} is incorrect!`;
```

    - Description: This is an example of functional cohesion, because the function is centered around 1 goal, creating a string for a failed login message.
- Type: logical

- 

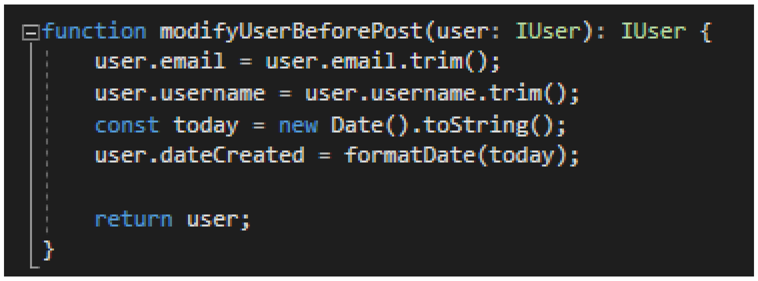
```csharp
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    if (!env.IsDevelopment())
    {
        app.UseSpaStaticFiles();
    }

    app.UseRouting();
    app.UseCors();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}");
    });

    app.UseSpa(spa =>
    {
        spa.Options.SourcePath = "ClientApp";

        if (env.IsDevelopment())
        {
            spa.UseProxyToSpaDevelopmentServer("http://localhost:8000/");
        }
    });
}
```

- Description: This “Configure” function demonstrates logical cohesion. The basis of the relationship between the things that happen in the “Configure” function is that they configure various different things unrelated to one another (REST API endpoints, SPA entry point, setting the development server port, etc.), but related to the web server as a whole.

- Type: informational
  - 

```csharp
public class User
{
    5 references
    public long ID { get; set; }
    6 references
    public string Username { get; set; }
    5 references
    public string Password { get; set; }
    7 references
    public string Email { get; set; }
}
```

  - Description: The User module represents informational cohesion because it has getters and setters (signified by “ { get; set; }”), which are used to read and update the properties for a given instance of the User class.
- Type: temporal

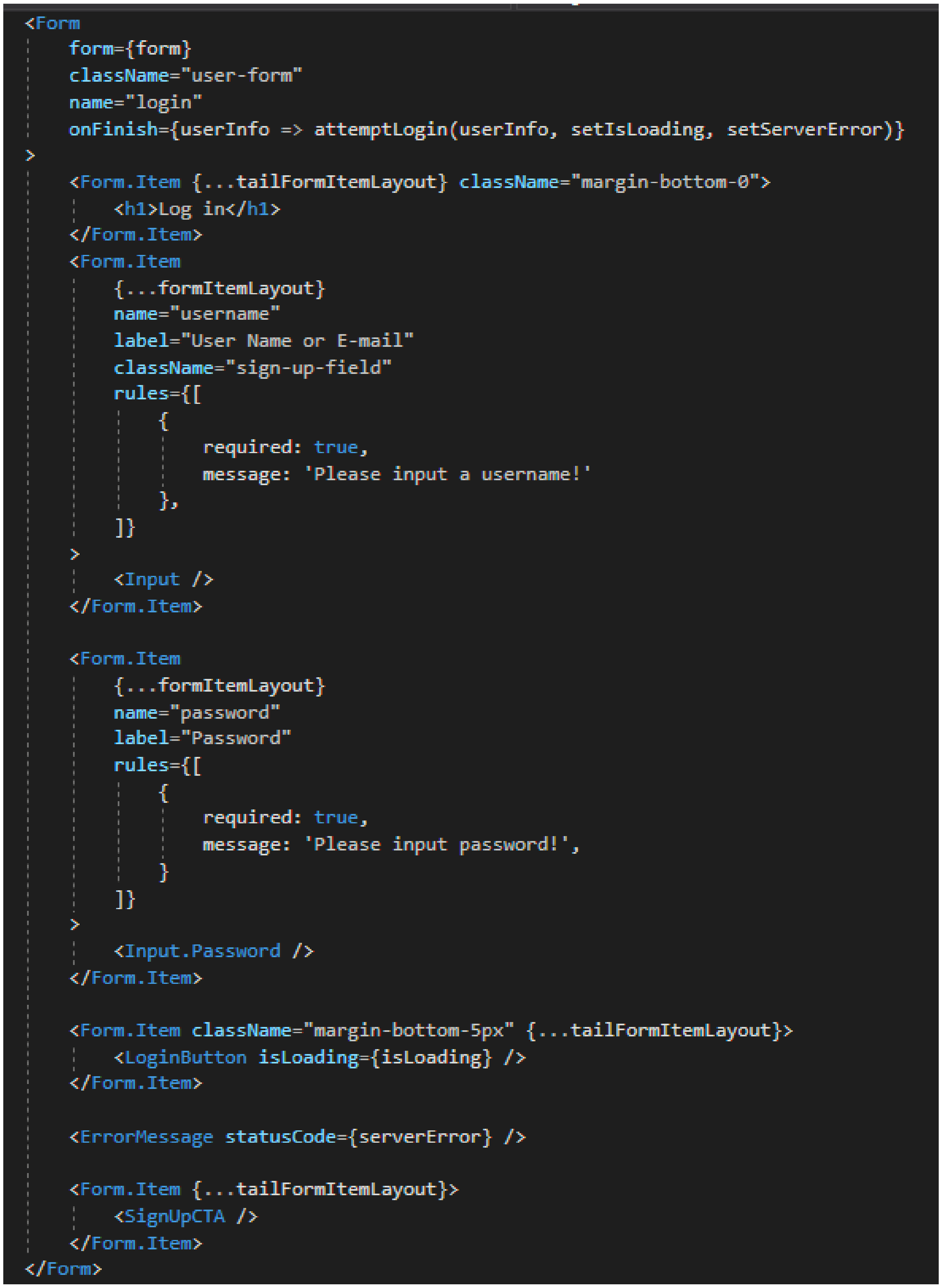
- 

```
function modifyUserBeforePost(user: IUser): IUser {
    user.email = user.email.trim();
    user.username = user.username.trim();
    const today = new Date().toString();
    user.dateCreated = formatDate(today);

    return user;
}
```

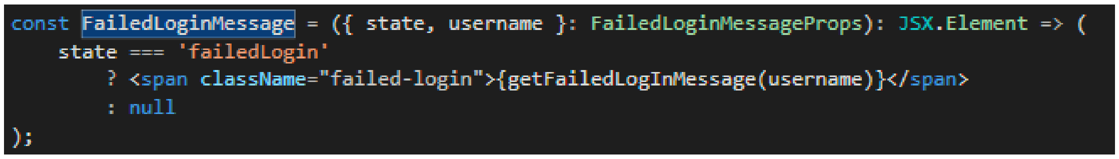
- Description: This function modifies the user to be in the correct state before sending it off the server to be saved to the database. Setting email, username, and date created could be their own standalone abstractions. Since some of the behavior is unrelated to one another (e.g adding the date property to a user, trimming the email, etc.) and the function is only used before user data is sent to the server, it has temporal cohesion.

Coupling

- Type: External

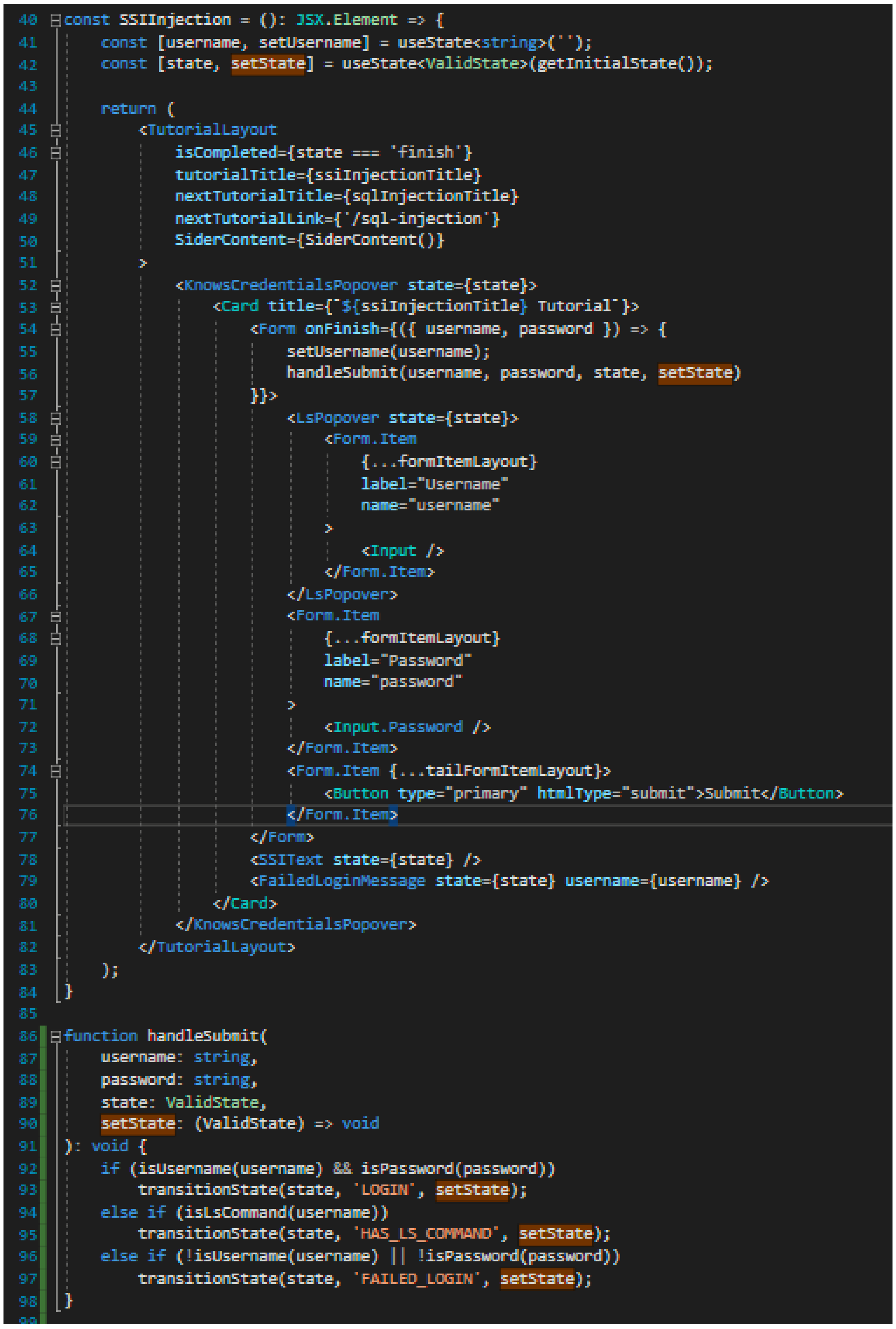


```
<Form
    form={form}
    className="user-form"
    name="login"
    onFinish={userInfo => attemptLogin(userInfo, setIsLoading, setServerError)}
>
    <Form.Item {...tailFormItemLayout} className="margin-bottom-0">
        <h1>Log in</h1>
    </Form.Item>
    <Form.Item
        {...formItemLayout}
        name="username"
        label="User Name or E-mail"
        className="sign-up-field"
        rules={[
            {
                required: true,
                message: 'Please input a username!'
            },
        ]}
    >
        <Input />
    </Form.Item>

    <Form.Item
        {...formItemLayout}
        name="password"
        label="Password"
        rules={[
            {
                required: true,
                message: 'Please input password!',
            }
        ]}
    >
        <Input.Password />
    </Form.Item>

    <Form.Item className="margin-bottom-5px" {...tailFormItemLayout}>
        <LoginButton isLoading={isLoading} />
    </Form.Item>

    <ErrorMessage statusCode={serverError} />

    <Form.Item {...tailFormItemLayout}>
        <SignUpCTA />
    </Form.Item>
</Form>
```

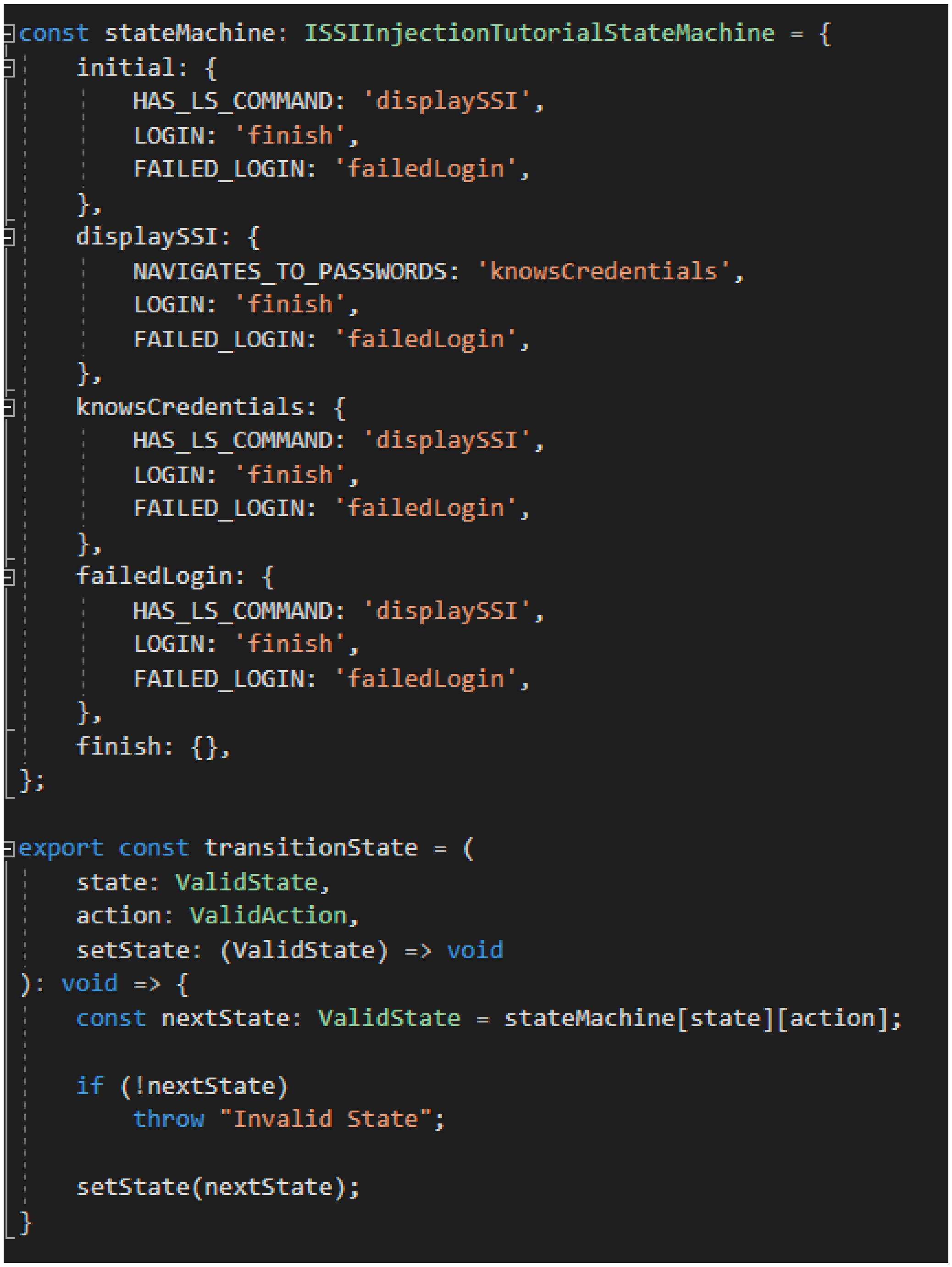
    - Description: We depend on an external component library, Ant Design, to allow us to move faster. In the above component we're depending on Form and Input from Ant Design.
- Type: control
    - 

```
const FailedLoginMessage = ({ state, username }: FailedLoginMessageProps): JSX.Element => (
    state === 'failedLogin'
        ? <span className="failed-login">{getFailedLogInMessage(username)}</span>
        : null
);
```

    - Description: The caller of this component passes it a state value and that conditionally determines what to render. If the login failed then it renders a message, otherwise it renders nothing.
- Type: control



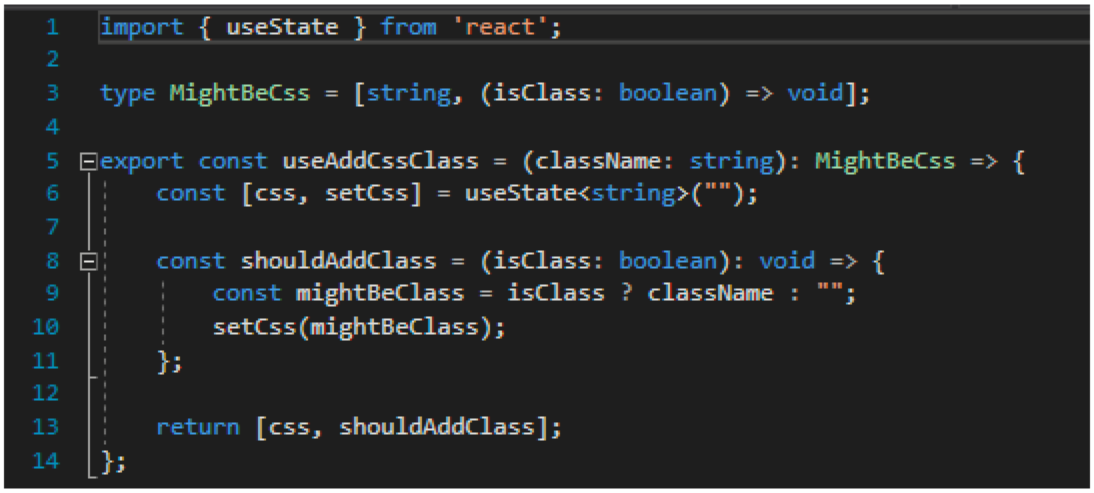
```
const SSIInjection = (): JSX.Element => {
    const [username, setUsername] = useState<string>('');
    const [state, setState] = useState<ValidState>(getInitialState());

    return (
        <TutorialLayout
            isCompleted={state === 'finish'}
            tutorialTitle={ssiInjectionTitle}
            nextTutorialTitle={sqlInjectionTitle}
            nextTutorialLink={'/sql-injection'}
            SiderContent={SiderContent()}
        >
            <KnowsCredentialsPopover state={state}>
                <Card title={`${ssiInjectionTitle} Tutorial`}>
                    <Form onFinish={({ username, password }) => {
                        setUsername(username);
                        handleSubmit(username, password, state, setState)
                    }}>
                        <LsPopover state={state}>
                            <Form.Item
                                {...formItemLayout}
                                label="Username"
                                name="username"
                            >
                                <Input />
                            </Form.Item>
                        </LsPopover>
                        <Form.Item
                            {...formItemLayout}
                            label="Password"
                            name="password"
                        >
                            <Input.Password />
                        </Form.Item>
                        <Form.Item {...tailFormItemLayout}>
                            <Button type="primary" htmlType="submit">Submit</Button>
                        </Form.Item>
                    </Form>
                    <SSIText state={state} />
                    <FailedLoginMessage state={state} username={username} />
                </Card>
            </KnowsCredentialsPopover>
        </TutorialLayout>
    );
}

function handleSubmit(
    username: string,
    password: string,
    state: ValidState,
    setState: (ValidState) => void
): void {
    if (isUsername(username) && isPassword(password))
        transitionState(state, 'LOGIN', setState);
    else if (isLsCommand(username))
        transitionState(state, 'HAS_LS_COMMAND', setState);
    else if (!isUsername(username) || !isPassword(password))
        transitionState(state, 'FAILED_LOGIN', setState);
}
```



```
const stateMachine: ISSIInjectionTutorialStateMachine = {
    initial: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    displaySSI: {
        NAVIGATES_TO_PASSWORDS: 'knowsCredentials',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    knowsCredentials: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    failedLogin: {
        HAS_LS_COMMAND: 'displaySSI',
        LOGIN: 'finish',
        FAILED_LOGIN: 'failedLogin',
    },
    finish: {},
};

export const transitionState = (
    state: ValidState,
    action: ValidAction,
    setState: (ValidState) => void
): void => {
    const nextState: ValidState = stateMachine[state][action];

    if (!nextState)
        throw "Invalid State";

    setState(nextState);
}
```

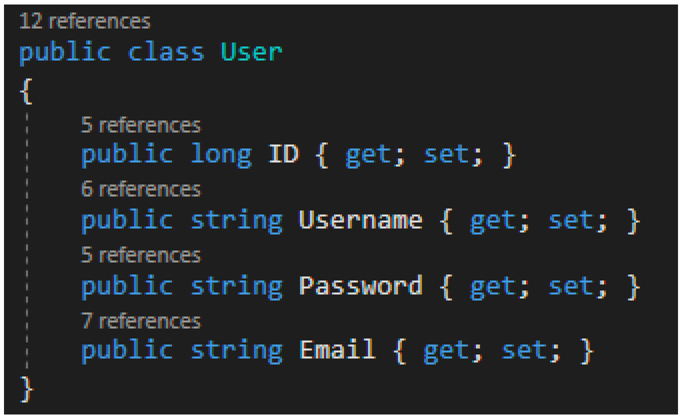
- Description: “setState” is a function that updates the “state” variable. In this example it’s being passed to a handle form submission function which then passes it off to a finite state machine to determine the next state of the program. The state machine transition function then updates the state of “state”.

- Type: data

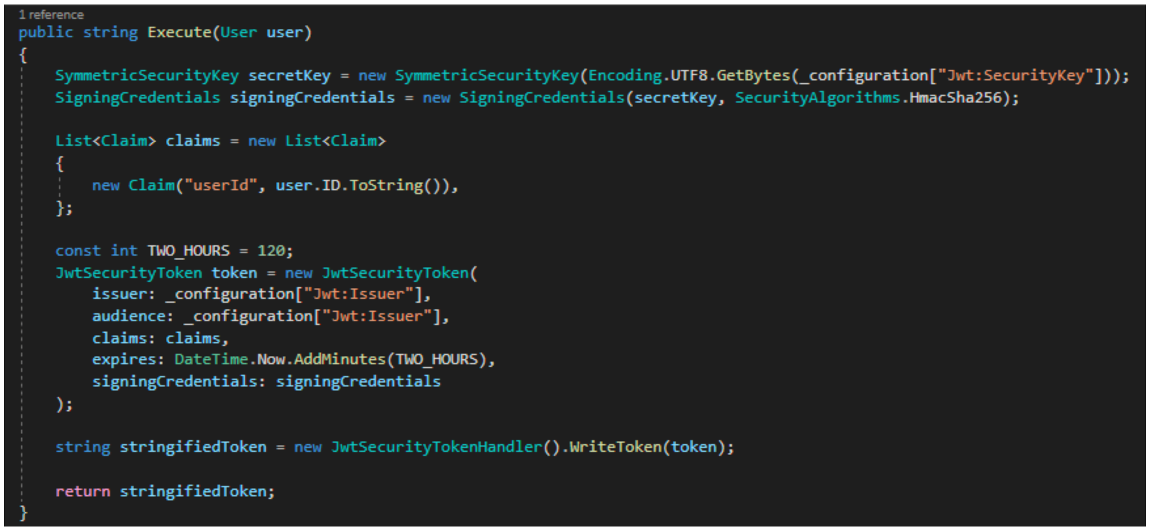
- 

```
import { useState } from 'react';

type MightBeCss = [string, (isClass: boolean) => void];

export const useAddCssClass = (className: string): MightBeCss => {
    const [css, setCss] = useState<string>("");

    const shouldAddClass = (isClass: boolean): void => {
        const mightBeClass = isClass ? className : "";
        setCss(mightBeClass);
    };

    return [css, shouldAddClass];
};
```

- Description: This module is a custom React hook to make CSS class changes more visible in components that use it. It just takes a string for the CSS class name and provides a variable that potentially holds the CSS class and a function to toggle that CSS class on and off. So, useAddCssClass is data coupled.

- Type: stamp
    - 

```
12 references
public class User
{
    5 references
    public long ID { get; set; }
    6 references
    public string Username { get; set; }
    5 references
    public string Password { get; set; }
    7 references
    public string Email { get; set; }
}
```

- 

```
1 reference
public string Execute(User user)
{
    SymmetricSecurityKey secretKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:SecurityKey"]));
    SigningCredentials signingCredentials = new SigningCredentials(secretKey, SecurityAlgorithms.HmacSha256);

    List<Claim> claims = new List<Claim>
    {
        new Claim("userId", user.ID.ToString()),
    };

    const int TWO_HOURS = 120;
    JwtSecurityToken token = new JwtSecurityToken(
        issuer: _configuration["Jwt:Issuer"],
        audience: _configuration["Jwt:Issuer"],
        claims: claims,
        expires: DateTime.Now.AddMinutes(TWO_HOURS),
        signingCredentials: signingCredentials
    );

    string stringifiedToken = new JwtSecurityTokenHandler().WriteToken(token);

    return stringifiedToken;
}
```

- Description: The “Execute” method has stamp coupling because it passes in a “User” object, but only uses the “ID” property. This is the case because in the future many properties of a user may be used for the JWT and JWT creation will always involve a user object.

Testing Report

Test Case ID	Test Case	Pre-Condition	Test Steps	Test Data	Expected Result	Actual Result	Status
Login_01	Can login with username.	User has created an account.	1. Navigate to Login page 2. Enter username 3. Enter password 4. Click "Login" button	<Valid username> <Valid associated password>	User is logged in.	User is logged in.	Pass
Login_02	Can login with email.	User has created an account.	1. Navigate to Login page 2. Enter email 3. Enter password 4. Click "Login" button	<valid email> <valid associated password>	User is logged in.	User is logged in.	Pass
Login_03	Cannot login with invalid credentials.	N/A	1. Navigate to Login page 2. Enter invalid username 3. Enter invalid password 4. Click "Login" button	<invalid username> <invalid password>	User does not get logged in.	User does not get logged in.	Pass
Login_04	Shows error message when logging in	N/A	1. Navigate to Login page 2. Enter invalid	<invalid username> <invalid passwor	Message that indicates to the user their	Message that indicates to the user	Pass

	with invalid credentials.		username 3. Enter invalid password 4. Click “Login” button	d>	login attempt has failed.	their login attempt has failed.	
Login_05	Give user visual indication that the system is trying to authenticate them.	N/A	1. Navigate to Login page 2. Enter username 3. Enter password 4. Click “Login” button	<username> <password>	A spinner appears inside the “Login” button.	A spinner appears inside the “Login” button.	Pass
Sign_Up_01	Disallows duplicate emails.	An account with the test email already exists.	1. Navigate to Sign Up page 2. Type a duplicate email in the email field	<email that already has an account associated with it>	Cannot sign up and appropriate message is shown.	Cannot sign up and appropriate message is shown.	Pass
Sign_Up_02	Disallows duplicate usernames.	An account with the test username already exists.	1. Navigate to Sign Up page 2. Type a duplicate username in the username field	<username that already has an account associated with it>	Cannot sign up and appropriate message is shown.	Cannot sign up and appropriate message is shown.	Pass
Sign_Up_03	Require all fields to be filled.	N/A	1. Navigate to Sign Up page 2. Click the “Sign Up” button	N/A	Cannot sign up and appropriate message is shown.	Cannot sign up and appropriate message is	Pass

						shown.	
Sign_Up_04	Can sign up when all sign up constraints are fulfilled.	N/A	1. Navigate to Sign Up page 2. Enter non-duplicate email 3. Enter non-duplicate username 4. Enter valid password 5. Enter password confirmation 6. Click the “Sign Up” button	<non-duplicate email> <non-duplicate username> password: Aa!12345678 password confirmation: Aa!12345678	User gets signed up.	User gets signed up.	Pass
Sign_Up_05	Correctly validates password.	N/A	1. Navigate to Sign Up page 2. Enter password without a symbol 3. Validate that the system says it’s incorrect 4. Enter password shorter than 8 characters 5. Validate that the system says it’s incorrect	Password without symbol: Aa12345678 Password without number: Aa!aaaaa Password without lowercase letter: AA!12345678 Password without	Password is only correct when it contains a symbol, uppercase letter, lowercase letter, number, and is 8 characters or larger. Shows appropriate error messages when applicable.	Password is only correct when it contains a symbol, uppercase letter, lowercase letter, number, and is 8 characters or larger. Shows appropriate error messages when applicabl	Pass

			6. Enter password without a number 7. Validate that the system says it’s incorrect 8. Enter password without a lowercase letter 9. Validate that the system says it’s incorrect 10. Enter password without an uppercase letter 11. Validate that the system says it’s incorrect 12. Enter password with a lowercase letter, uppercase letter, symbol, number, and 8+ character length 13. Validate	uppercase letter: aa!12345678 Password shorter than 8 characters: Aa!1		e.	

			that it's passes validation				
SSI_Injection_01	Display ls command initially.	User is logged in.	1. Navigate to the SSI Injection tutorial page	N/A	Popover appears indicating where the user can enter the ls command and why it's relevant.	Popover appears indicating where the user can enter the ls command and why it's relevant.	Pass
SSI_Injection_02	Display ssi text after entering ls command.	User is logged in.	1. Navigate to the SSI Injection tutorial page 2. Enter the ls command into the username field as instructed 3. Submit the login form	ls command: "<!--#exec cmd="ls ../../"-->"	Text appears in the failed authentication message area relating to items exposed by the SSI Injection. An associated popover explaining what it is and what to do also appears.	Text appears in the failed authentication message area relating to items exposed by the SSI Injection. An associated popover explaining what it is and what to do also appears.	Pass
SSI_Injection_03	Display knows credentials	The user is logged in and	1. Navigate back to SSI	N/A	Displays message indicating	Displays message indicating	Pass

	message after they've visited the secret file.	has seen the files exposed by the SSI Injection.	Injection tutorial page		to the user to enter the credentials they found in passwords.txt.	g to the user to enter the credentials they found in passwords.txt.	
SSI_Injection_04	Display error message when entering incorrect credentials.	The user is logged in.	1. Navigate to the SSI Injection tutorial page 2. Enter invalid username 3. Enter invalid password	Invalid username: 123 Invalid password: 321	User does not complete the tutorial and an appropriate error message is displayed.	User does not complete the tutorial and an appropriate error message is displayed.	Pass
SSI_Injection_05	Display congrats page when correct credentials are submitted.	The user is logged in.	1. Navigate to the SSI Injection tutorial page 2. Enter valid username 3. Enter valid password	Valid username: "admin" Valid password: "Totally-Not-Hackable-Password!"	User completes the tutorial and is shown the congrats page.	User completes the tutorial and is shown the congrats page.	Pass
SQL_Injection_01	Initially display a username entry message.	User is logged in.	1. Navigate to the SQL Injection tutorial page	N/A	A popover is displayed indicating to the user to type the admin username in the	A popover is displayed indicating to the user to type the admin	Pass

					username field.	email in the username field.	
SQL_Injection_02	Display SQL Injection message after user enters correct username.	User is logged in.	1. Navigate to the SQL Injection tutorial page 2. Type admin email into the username field	Admin username: “admin”	A popover is displayed indicating to the user to type in the SQL command that allows for SQL Injection in the password field.	A popover is displayed indicating to the user to type in the SQL command that allows for SQL Injection in the password field.	Pass
SQL_Injection_03	Advise the user to click the “Log In” button after setting up the attack.	User is logged in.	1. Navigate to the SQL Injection tutorial 2. Enter the admin username in the username field 3. Enter the SQL Injection attack into the password field	Admin username: “admin” SQL Injection attack: “‘ OR 1 = 1”	A popover appears indicating to the user that they have the attack setup and just need to click “Log In”.	A popover appears indicating to the user that they have the attack setup and just need to click “Log In”.	Pass
SQL_Injection_04	Display error message	User is logged in.	1. Navigate to the SQL Injection	Invalid username:	Error message appears	Error message appears	Pass

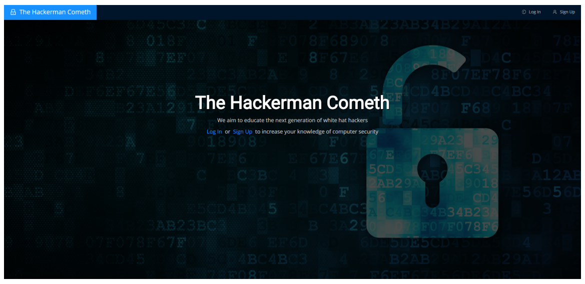
	when incorrect credentials are submitted. User does not complete the tutorial with these credentials.		tutorial page 2. Type in an invalid username in username field 3. Type in invalid password in password field 4. Click the “Log In” button	“invalid username” Invalid password: “invalid password”	indicating that the username /password combination is not valid.	indicating that the username/password combination is not valid.	
SQL_Injection_05	Display congrats page when correct credentials are submitted.	User is logged in.	1. Navigate to the SQL Injection tutorial page. 2. Type in correct username in username field 3. Type in correct password in password field 4. Click the “Log In” button	Correct username: “admin” Correct password: “‘ OR 1 = 1”	The congrats page for the SQL Injection tutorial appears.	The congrats page for the SQL Injection tutorial appears.	Pass
Sign_Out_01	When signed out redirects to login page.	User is logged in.	1. Click “Sign Out” link in navbar	N/A	User is redirected to the login page.	User is redirected to the login page.	Pass

Sign_Out_02	When signed out token is removed from local storage.	User is logged in.	1. Click "Sign Out" link in navbar 2. Open the DevTools for your browser	N/A	The JSON Web Token is no longer in local storage.	The JSON Web Token is no longer in local storage.	Pass
Sign_Out_03	When signed out user can no longer access SSI Injection tutorial.	User was logged in and just signed out.	1. Navigate to ~/app/ssi-injection	N/A	User cannot access the SSI Injection tutorial and is redirected to the login page.	User cannot access the SSI Injection tutorial and is redirected to the login page.	Pass
Sign_Out_04	When signed out user can no longer access SQL Injection tutorial.	User was logged in and just signed out.	1. Navigate to ~/app/sql-injection	N/A	User cannot access the SQL Injection tutorial and are redirected to the login page.	User cannot access the SQL Injection tutorial and are redirected to the login page.	Pass
Sign_Out_05	When logged in Sign Out appears on the navbar.	N/A.	1. Log in	N/A	A "Sign Out" link appears in the navbar.	A "Sign Out" link appears in the navbar.	Pass
Sign_out_06	When signed out "Sign Out" does not appear on	User is logged in.	1. Click the "Sign Out" link in the navbar	N/A	The "Sign Out" link no longer appears in the	The "Sign Out" link no longer	Pass

	the navbar.				navbar.	appears in the navbar.	

Demo Screenshots

Unauthenticated Landing Page



The Hackerman Cometh

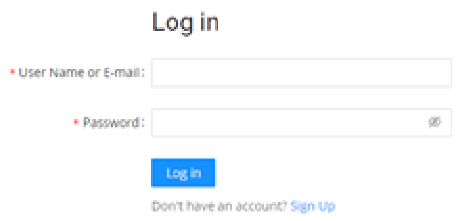
We aim to educate the next generation of white hat hackers

Log In or Sign Up to increase your knowledge of computer security

The first page that a user sees on the website contains the name of the website and a call to action to either log in or sign up to increase their knowledge of computer security. “Log In” and “Sign Up” are links that take the user to the associated page. At the top of the page you will notice the Navigation bar. The Navigation bar has two states one prior to a logging in and one after.

The Navigation bars unauthenticated state (prior to a log in) has three buttons. The Hackerman Cometh button brings you to the landing page, the login button brings you to the Log In page and the Sign up button brings you to the sign up page.

Log in


Log in

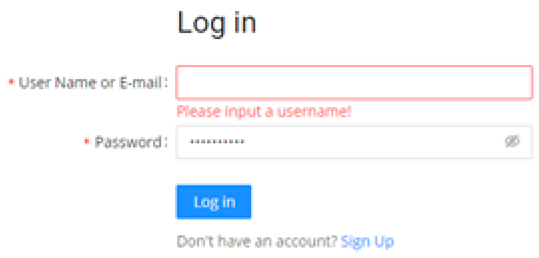
* User Name or E-mail:

* Password:

Log in

Don't have an account? Sign Up

The log in page contains two input boxes with required fields. The first input box for “User Name or E-mail” accepts either the username or the email that the user registered with when setting up their account.


Log in

* User Name or E-mail:

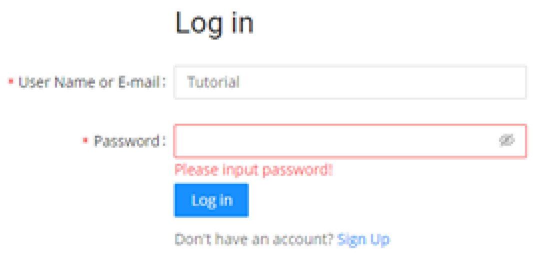
Please input a username!

* Password: ••••••••••

Log in

Don't have an account? Sign Up

If the user leaves the “User Name or E-mail” field empty the input will display an error when the user clicks “Log In”.


Log in

* User Name or E-mail: Tutorial

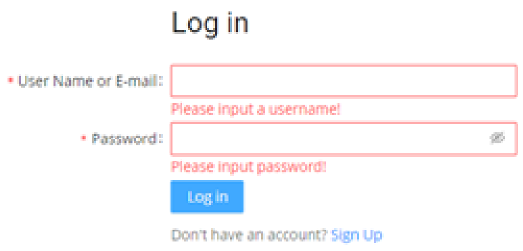
* Password:

Please input password!

Log in

Don't have an account? Sign Up

If the user leaves the “Password” field empty the input will display an error when the user clicks “Log In”.


Log in

* User Name or E-mail:

Please input a username!

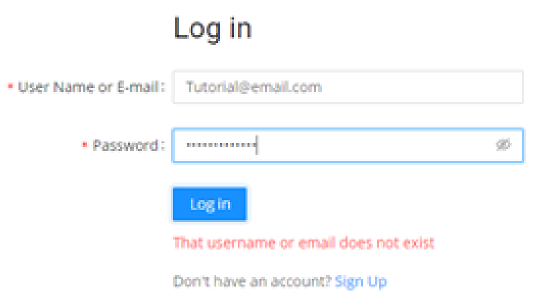
* Password:

Please input password!

Log in

Don't have an account? Sign Up

If neither “User Name or E-mail” nor “Password” fields are filled out both input boxes display errors when the user clicks “Log In”.



Log in

* User Name or E-mail: Tutorial@email.com

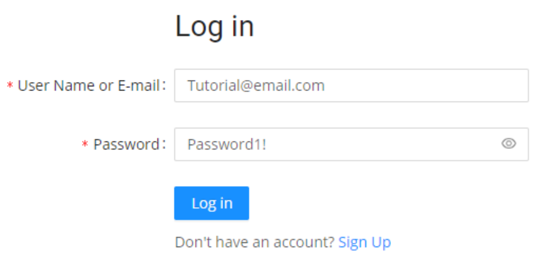
* Password: ••••••••••••••

Log in

That username or email does not exist

Don't have an account? Sign Up

If the user does not enter a valid “User Name or E-mail” the system will display an error to the user that the username they entered does not exist.



Log in

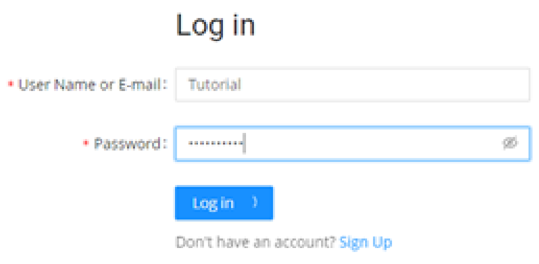
* User Name or E-mail: Tutorial@email.com

* Password: Password1!

Log in

Don't have an account? Sign Up

The password is hidden as the user types, but the input box has the feature on its right that looks like an eye; when pressed it allows the user to hide or unhide the password as it is being typed.



Log in

* User Name or E-mail: Tutorial

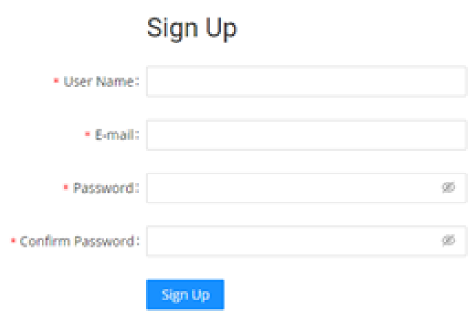
* Password: ••••••••••

Log in

Don't have an account? Sign Up

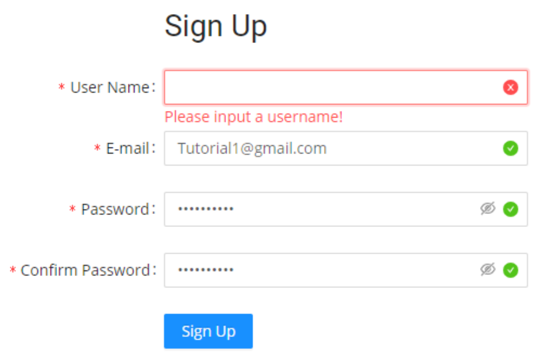
Upon a successful log in attempt the “Log In” button displays a loading icon until the user is logged in.

Sign up


Sign Up

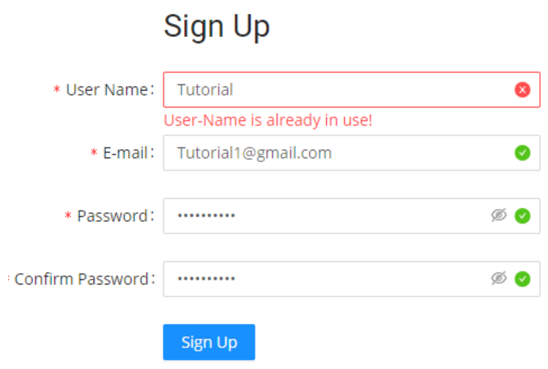
* User Name:
* E-mail:
* Password:
* Confirm Password:

Sign up consists of four required fields and a button to sign up.

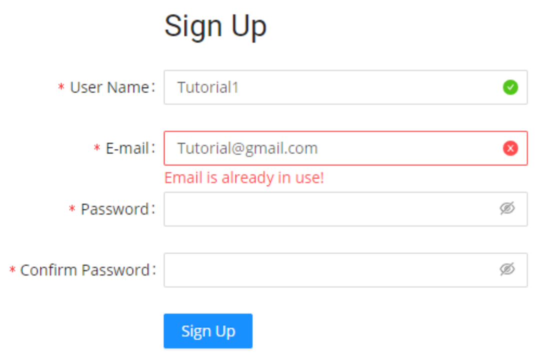

Sign Up

* User Name:
Please input a username!
* E-mail: Tutorial1@gmail.com
* Password: ••••••••••
* Confirm Password: ••••••••••

Each input displays an error if it is not filled out when the “Sign Up” button is clicked.

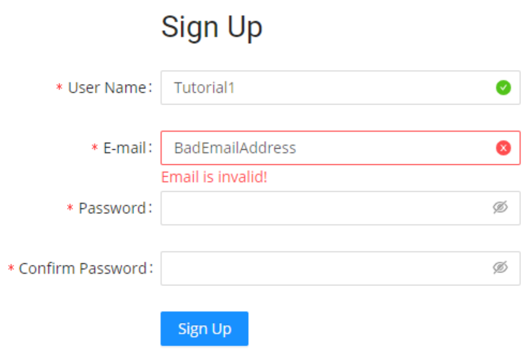

Sign Up

* User Name: Tutorial
User-Name is already in use!
* E-mail: Tutorial1@gmail.com
* Password: ••••••••••
Confirm Password: ••••••••••


Sign Up

* User Name: Tutorial1
* E-mail: Tutorial@gmail.com
Email is already in use!
* Password:
* Confirm Password:

The system will also detect if a username or email is already in use and will alert the user in these cases. Every user must have a unique username and email.


Sign Up

* User Name: Tutorial1

* E-mail: BadEmailAddress

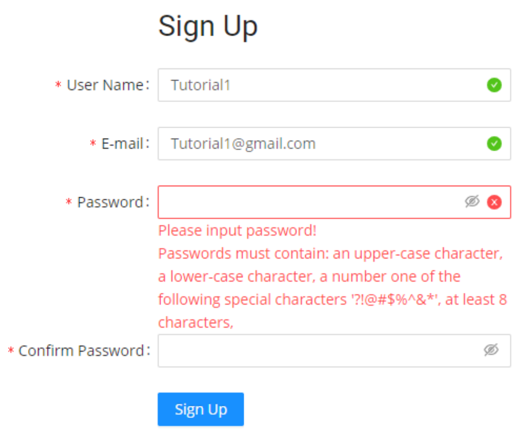
Email is invalid!

* Password:

* Confirm Password:

Sign Up

In addition to determining if an email is already in use the system also detects if the text in the email field is in fact a valid email. If the email is invalid the user is alerted.


Sign Up

* User Name: Tutorial1

* E-mail: Tutorial1@gmail.com

* Password:

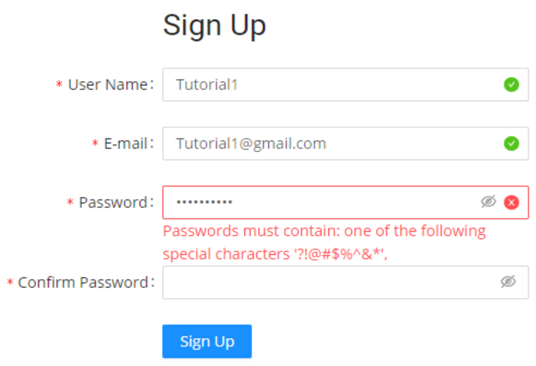
Please input password!

Passwords must contain: an upper-case character, a lower-case character, a number one of the following special characters '?!@#$%^&*', at least 8 characters,

* Confirm Password:

Sign Up

The Password field contains user requirements for password complexity.


Sign Up

* User Name: Tutorial1

* E-mail: Tutorial1@gmail.com

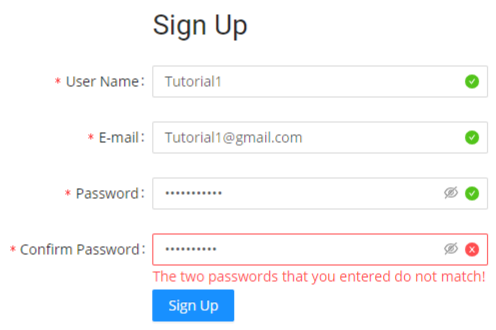
* Password: ••••••••••

Passwords must contain: one of the following special characters '?!@#$%^&*',

* Confirm Password:

Sign Up

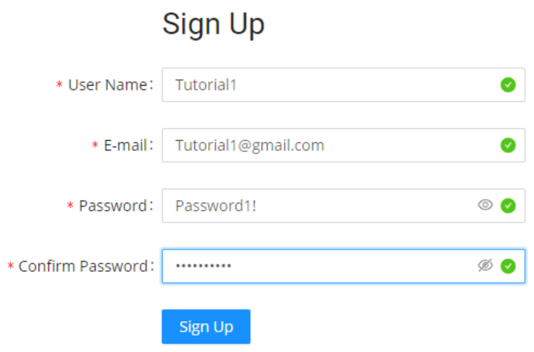
As the user types in their password, the system will update them based on what requirements are needed. As the requirements are met, the system alerts will disappear. When all requirements have been met a green check mark will appear.


Sign Up

* User Name:	Tutorial1 ✓
* E-mail:	Tutorial1@gmail.com ✓
* Password:	••••••••••• ✓
* Confirm Password:	•••••••••• ✗

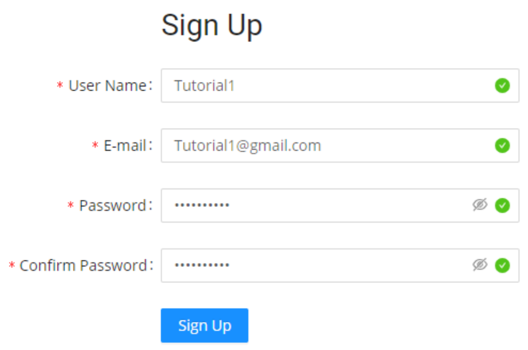
The two passwords that you entered do not match!

The confirm password field must match that of the password field. If they do not match the user will be alerted and will not be allowed to register until the passwords match.


Sign Up

* User Name:	Tutorial1 ✓
* E-mail:	Tutorial1@gmail.com ✓
* Password:	Password1! ✓
* Confirm Password:	•••••••••• ✓

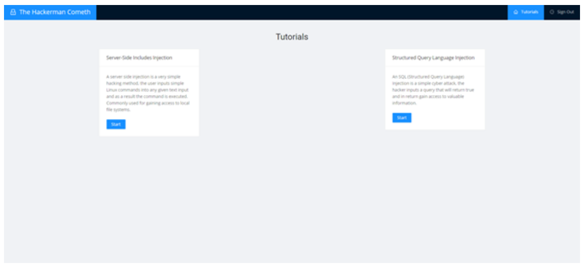
In addition to this, the “Password” and “Confirm Password” fields have a feature to allow you to hide/unhide the text in the input box by clicking on the eye icon on the right side of the input box.


Sign Up

* User Name:	Tutorial1 ✓
* E-mail:	Tutorial1@gmail.com ✓
* Password:	•••••••••• ✓
* Confirm Password:	•••••••••• ✓

If the information in every field is valid then every input box will contain a green check mark and the user will be able to click the “Sign Up” button to register.

Tutorial Selection Page



Tutorials

Server-Side Includes Injection

A server side injection is a very simple hacking method, the user inputs simple Linux commands into any given text input and as a result the command is executed. Commonly used for gaining access to local file systems.

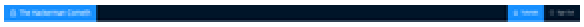
Start

Structured Query Language Injection

An SQL (Structured Query Language) injection is a simple cyber attack, the hacker inputs a query that will return true and in return gain access to valuable information.

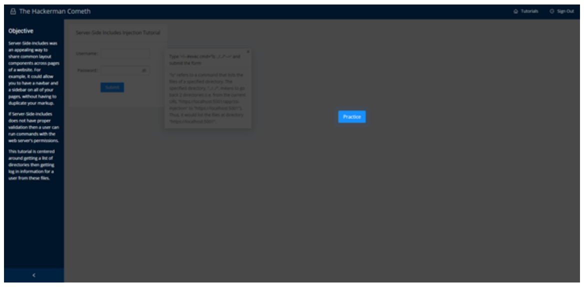
Start

The tutorial selection menu contains all the tutorials on the website. It gives a brief overview of what you will learn in each tutorial, as well as buttons that allow you to begin the associated tutorial.



After a successful sign in has occurred the navigation bar changes, in lieu of the “Log In” and “Sign Up” buttons the user will now see a “Tutorials” button and a “Sign Out” button. Clicking the “Tutorials” or “The Hackerman Cometh” button will bring the user to the Tutorial selection page and the button will remain highlighted blue so long as the user stays on the Tutorial selection page. The Sign Out button will sign the user out of the system, revoking their access to tutorials, and bringing them to the Log In page.

Tutorial Layout



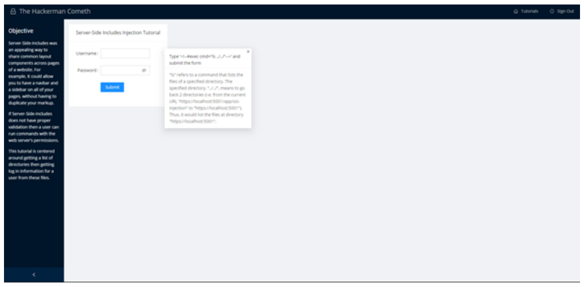
Objective

Server-Side includes was an appealing way to share common layout components across pages of a website. For example, it could allow you to have a navbar and a sidebar on all of your pages, without having to duplicate your markup.

If Server-Side includes does not have proper validation then a user can run commands with the web server's permissions.

This tutorial is centered around getting a list of directories then getting log in information for a user from these files.

Each tutorial has an objective on the left hand side of the screen. The objective gives the user insight on how to approach each tutorial as well as any other information that would be helpful to them. The tutorial itself is greyed out until the user clicks the "Practice" button, which allows them to start the tutorial.



Objective

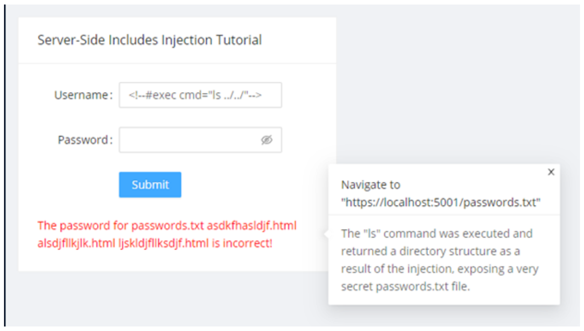
Server-Side includes was an appealing way to share common layout components across pages of a website. For example, it could allow you to have a navbar and a sidebar on all of your pages, without having to duplicate your markup.

If Server-Side includes does not have proper validation then a user can run commands with the web server's permissions.

This tutorial is centered around getting a list of directories then getting log in information for a user from these files.

Type "<!--#exec cmd="ls ../../"-->" and submit the form

"ls" refers to a command that lists the files of a specified directory. The specified directory, "../../", means to go back 2 directories (i.e. from the current URL "https://localhost:5001/app/ssi-injection" to "https://localhost:5001"). Thus, it would list the files at directory "https://localhost:5001".



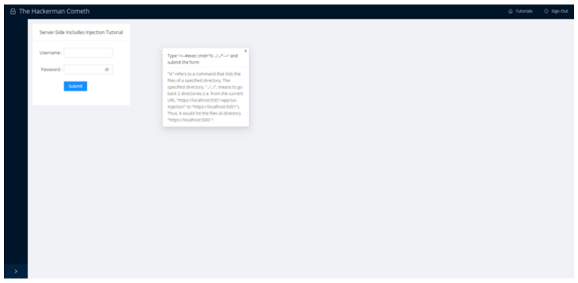
<!--#exec cmd="ls ../../"-->

The password for passwords.txt asdkfhasldjf.html alsdjfllkjlk.html ljskldjflksdjf.html is incorrect!

Navigate to "https://localhost:5001/passwords.txt"

The "ls" command was executed and returned a directory structure as a result of the injection, exposing a very secret passwords.txt file.

Inside the tutorial there are popovers, which indicate an action the user will perform and a description as to why they are performing that action.

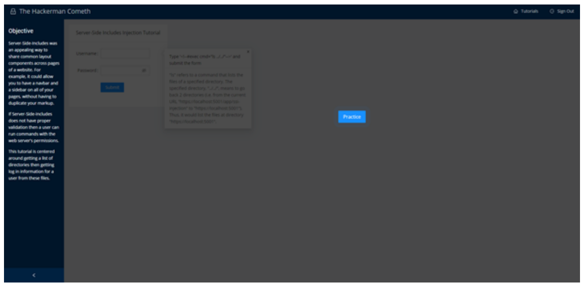


Type "<!--#exec cmd="ls ../../"-->" and submit the form

"ls" refers to a command that lists the files of a specified directory. The specified directory, "../../", means to go back 2 directories (i.e. from the current URL "https://localhost:5001/app/ssi-injection" to "https://localhost:5001"). Thus, it would list the files at directory "https://localhost:5001".

The objective menu also contains a collapse feature. The arrow on the bottom of the objective allows the user to collapse and uncollapse the objective.

Server-Side Includes (SSI) Injection Tutorial



Objective

Server-Side includes was an appealing way to share common layout components across pages of a website. For example, it could allow you to have a navbar and a sidebar on all of your pages, without having to duplicate your markup.

If Server-Side includes does not have proper validation then a user can run commands with the web server's permissions.

This tutorial is centered around getting a list of directories then getting log in information for a user from these files.

Server-Side Includes Injection Tutorial

Username:

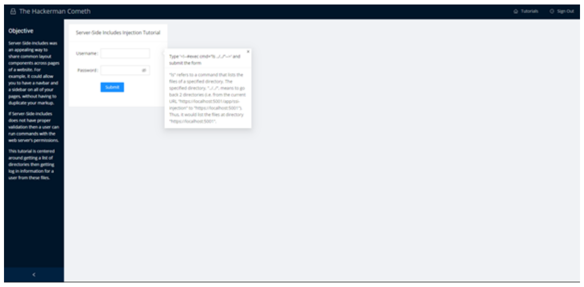
Password:

Submit

Type '<!--#exec cmd="ls ../../"-->' and submit the form

"ls" refers to a command that lists the files of a specified directory. The specified directory, "../../", means to go back 2 directories (i.e. from the current URL "https://localhost:5001/app/ssi-injection" to "https://localhost:5001"). Thus, it would list the files at directory "https://localhost:5001".

The objective tells the user what SSI is and what it is used for and why it is useful to know for cyber security reasons.



Objective

Server-Side includes was an appealing way to share common layout components across pages of a website. For example, it could allow you to have a navbar and a sidebar on all of your pages, without having to duplicate your markup.

If Server-Side includes does not have proper validation then a user can run commands with the web server's permissions.

This tutorial is centered around getting a list of directories then getting log in information for a user from these files.

Server-Side Includes Injection Tutorial

Username:

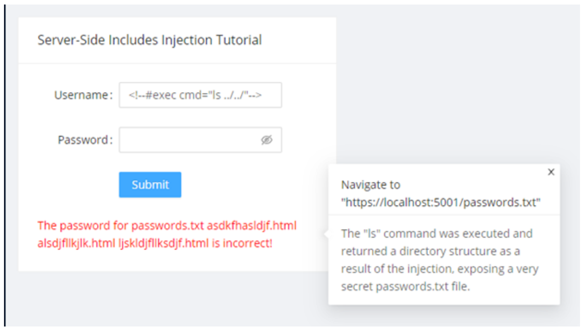
Password:

Submit

Type '<!--#exec cmd="ls ../../"-->' and submit the form

"ls" refers to a command that lists the files of a specified directory. The specified directory, "../../", means to go back 2 directories (i.e. from the current URL "https://localhost:5001/app/ssi-injection" to "https://localhost:5001"). Thus, it would list the files at directory "https://localhost:5001".

The popover tells the user what command they should enter and what the command does. It describes why the user is entering the command and the expected result.



Server-Side Includes Injection Tutorial

Username: <!--#exec cmd="ls ../../"-->

Password:

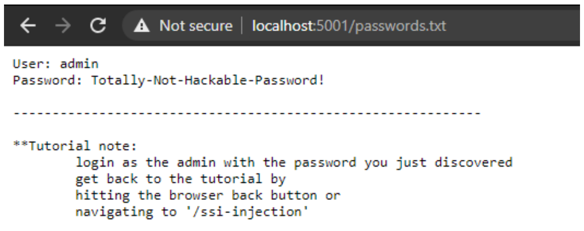
Submit

The password for passwords.txt asdkfhasldjf.html alsdjfllkjlk.html ljskldjfllksdjf.html is incorrect!

Navigate to "https://localhost:5001/passwords.txt"

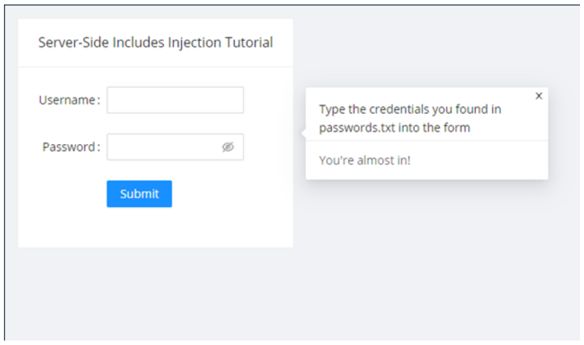
The "ls" command was executed and returned a directory structure as a result of the injection, exposing a very secret passwords.txt file.

Once the user submits the form, with the injection in place, they can see the result of the injection below the “Submit” button (the red text). The popover describes what the red text means and instructs the user to navigate to the “passwords.txt” file to find the credentials.



```
User: admin
Password: Totally-Not-Hackable-Password!

------------------------------------------------------------

**Tutorial note:
        login as the admin with the password you just discovered
        get back to the tutorial by
        hitting the browser back button or
        navigating to '/ssi-injection'
```

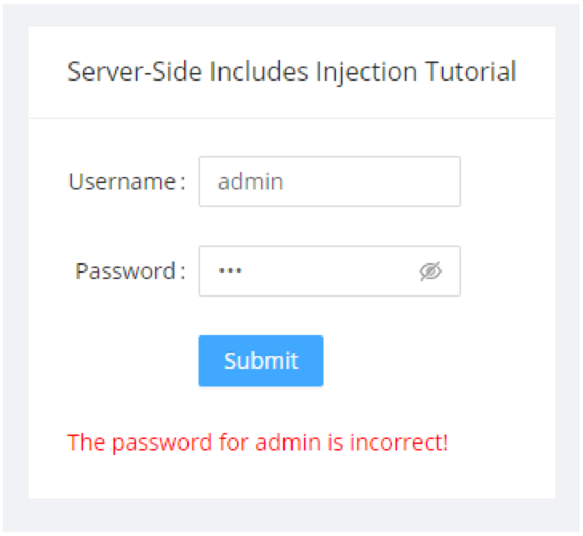
When the user navigates to the "passwords.txt" file they discover the "User" and "Password" for admin access and are also instructed how to return to the SSI Injection tutorial and complete the attack.



Type the credentials you found in passwords.txt into the form

You're almost in!

Upon Navigating back to the SSI Injection Tutorial the popover instructs them to enter the credentials they just discovered through the attack in "passwords.txt".

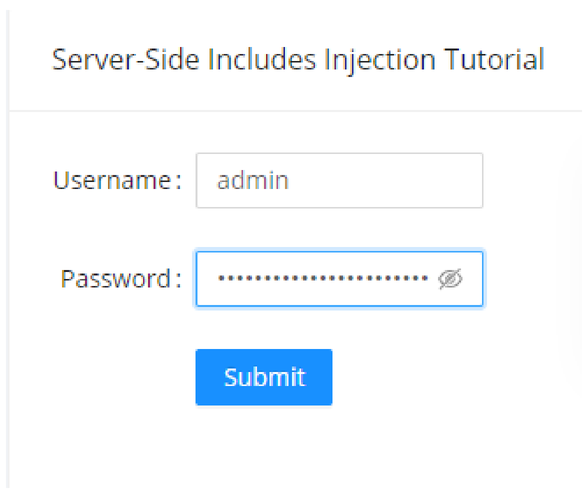


Username : admin

Password : •••

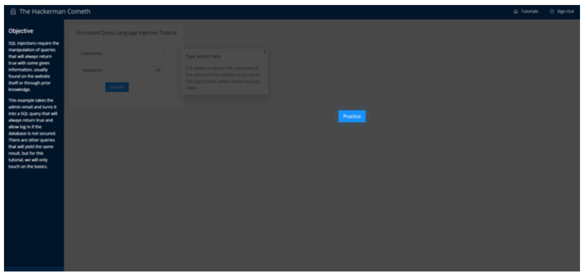
The password for admin is incorrect!

If the user enters the wrong credentials they will be barred from continuing on in the tutorial.



Submitting the form with the correct credentials from “passwords.txt” allows the user to complete the tutorial. They are then brought to the tutorial completion page.

Structured Query Language (SQL) Injection Tutorial

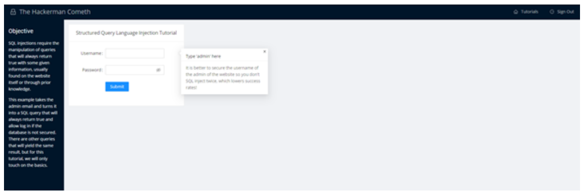


Objective

SQL injections require the manipulation of queries that will always return true with some given information, usually found on the website itself or through prior knowledge.

This example takes the admin email and turns it into a SQL query that will always return true and allow log in if the database is not secured. There are other queries that will yield the same result, but for this tutorial, we will only touch on the basics.

The objective describes what a SQL injection is, why it's relevant, and how it will be used in the tutorial.



Objective

SQL injections require the manipulation of queries that will always return true with some given information, usually found on the website itself or through prior knowledge.

This example takes the admin email and turns it into a SQL query that will always return true and allow log in if the database is not secured. There are other queries that will yield the same result, but for this tutorial, we will only touch on the basics.

Structured Query Language Injection Tutorial

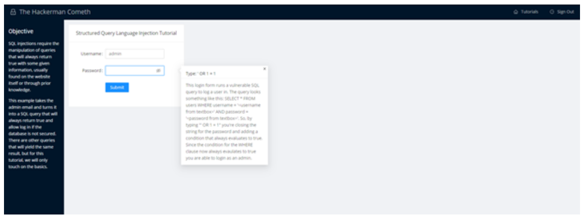
Username:

Password:

Type 'admin' here

It is better to secure the username of the admin of the website so you don't SQL inject twice, which lowers success rates!

The popover instructs the user to enter the username that the attack will be performed on, which is the admin.



Objective

SQL injections require the manipulation of queries that will always return true with some given information, usually found on the website itself or through prior knowledge.

This example takes the admin email and turns it into a SQL query that will always return true and allow log in if the database is not secured. There are other queries that will yield the same result, but for this tutorial, we will only touch on the basics.

Structured Query Language Injection Tutorial

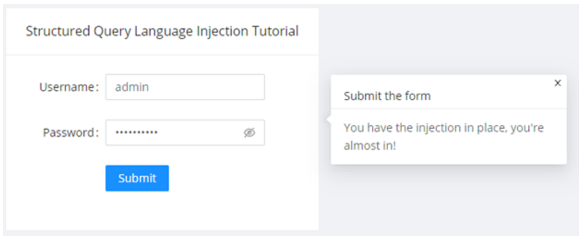
Username: admin

Password:

Type: ' OR 1 = 1

This login form runs a vulnerable SQL query to log a user in. The query looks something like this: SELECT * FROM users WHERE username = '<username from textbox>' AND password = '<password from textbox>'. So, by typing "' OR 1 = 1" you're closing the string for the password and adding a condition that always evaluates to true. Since the condition for the WHERE clause now always evaluates to true you are able to login as an admin.

After the user types in the admin username the next popover instructs the user to enter the SQL query to be injected. The popover also describes how the injection works and why the user types the query that they did.



Structured Query Language Injection Tutorial

Username: admin

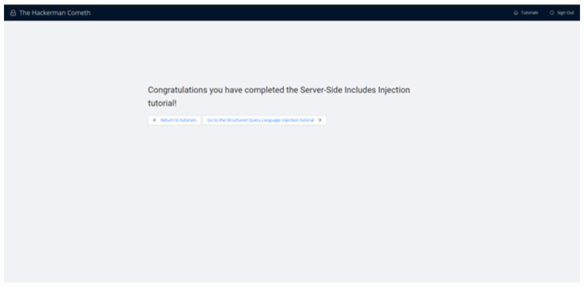
Password: ••••••••••

Submit the form

You have the injection in place, you're almost in!

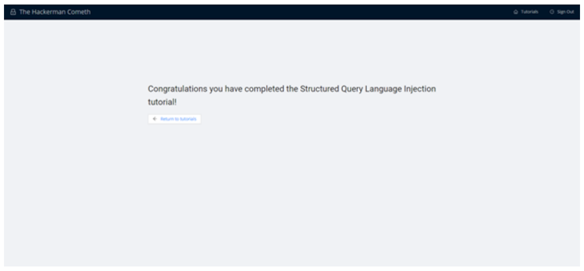
Once the user has entered the query to be injected the next popover instructs them to submit the form to perform the SQL Injection attack and complete the tutorial. They are then brought to the tutorial completion page.

Tutorial Completion Page



Congratulations you have completed the Server-Side Includes Injection tutorial!

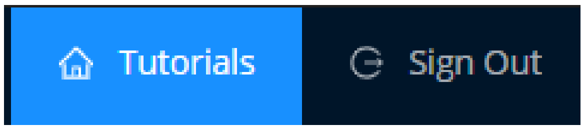
Upon successful injection of the attack, the user is directed to the tutorial completion page which gives them the option to move to the next tutorial or proceed back to the tutorial selection menu.

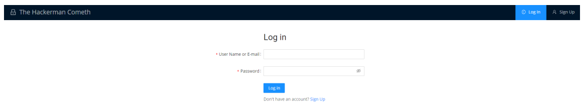


Congratulations you have completed the Structured Query Language Injection tutorial!

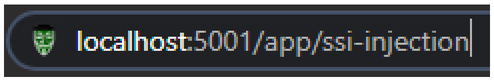
When you complete the most recent tutorial, the option to go to the next tutorial is not displayed and instead only the button to go back to the tutorial selection is visible.

Sign out

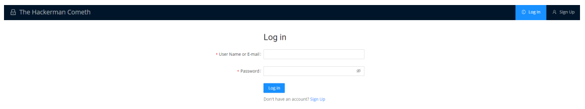




When the “Sign Out” button is clicked the user loses access to tutorials (authenticated routes) and is redirected to the Log In page. Once they log back in they can access the tutorials.



localhost:5001/app/ssi-injection



If the user tries to navigate to a tutorial (through a URL, since they can longer access them through the UI) after they have signed out they will be redirected to the Log In page since they are not authenticated. They must log in to gain access to the tutorials.

Tools

- Requirements
    - Use Case Models
- Prototype
    - Figma
- Design Tools
    - draw.io
- Development
    - C#
    - ASP.NET Core
    - Entity Framework Core
    - SQL Server
    - JavaScript
    - TypeScript
    - GatsbyJS
    - ReactJS
    - Ant Design
    - GraphQL
    - SCSS
    - Yarn
    - Nuget
    - Visual Studio
    - Git
- Process
    - Azure DevOps (houses our Kanban board, wiki, and repo)

Read Me

Installation

- Install Visual Studio 2019 Community
- Install npm
- Install yarn
- Run npm i -g gatsby-cli
- Run git clone https://CS360@dev.azure.com/CS360/The-Hackerman-Cometh/_git/The-Hackerman-Cometh when prompted for a password use “kvpxmek2js5vycyddr6myie6ewo6qnmqwesquvllojpxy3zogssq”
- In the Web project navigate to the ClientApp directory and run yarn

Run in Development Mode

- In the Web project navigate to the ClientApp directory and run yarn develop
- In Visual Studio Navigate to Debug -> Start Without Debugging (Ctrl+F5)

Run in Production Mode

- In the Web project navigate to the ClientApp directory and run yarn build
- In the Web project navigate to the ClientApp directory and run yarn serve -p 8000
- In Visual Studio Navigate to Debug -> Start Without Debugging (Ctrl+F5)

References

https://pmtips.net/article/defining-project-constraints
https://www.scaledagileframework.com/nonfunctional-requirements/
https://en.wikipedia.org/wiki/Single-page_application
https://www.crazyegg.com/blog/speed-up-your-website/
https://gregberge.com/blog/gatsby-seo
https://pmtips.net/article/defining-project-constraints
https://www.scaledagileframework.com/nonfunctional-requirements/
https://en.wikipedia.org/wiki/Single-page_application
https://www.crazyegg.com/blog/speed-up-your-website/
https://gregberge.com/blog/gatsby-seo
https://css-tricks.com/robust-react-user-interfaces-with-finite-state-machines/
https://jwt.io/
https://reactjs.org/
https://www.gatsbyjs.org/docs/
https://www.professionalqa.com/cohesion-in-software-testing