UT08.3: Funcionalidades Mitienda

Contenido

1. Introducción

Vamos a crear una serie de funcionalidades basadas en la gestión de cookies y en el api `WebStorage`. Las funcionalidades que vamos a implementar son:

- Mensaje de inicio para indicar que nuestra web utiliza cookies.
- Servicio de autenticación que permita mostrar contenido en base si estamos autenticados o no.
- Mantener las preferencias del usuario, en este caso para elegir el idioma de la página.

Estas funcionalidades no tendrán una implementación real al no disponer de un servidor, pero nos permitirá jugar con los dos mecanismos de almacenamiento que hemos visto en esta unidad.

2. Mensaje de uso de cookies

Vamos a mostrar un mensaje de que nuestra aplicación está haciendo uso de cookies. Este mensaje debe mostrarse al usuario como primera acción de la aplicación mediante una notificación del Bootstrap denominada *toast*. En el caso de aceptar el mensaje, la notificación se cerrará y se tendrá acceso al resto de funciones de la aplicación. La próxima vez que se acceda a la aplicación, el mensaje no debe aparecer porque ya se han aceptado las condiciones.

Para detectar que el usuario ha aceptado las condiciones, debemos dejar una cookie, la cual vamos a denominar `accetedCookieMessage`, cuando se cliquee en el botón de aceptar de la notificación. En el inicio de la página debemos leer si existe dicha cookie para mostrar o el modal al inicio de la aplicación.

2.1. Módulo util.js

Vamos a crear un módulo con las funciones genéricas que se puedan utilizar en diferentes puntos de la aplicación. De momento contendrá las funciones para obtener y dejar una cookie para facilitar su acceso dado lo tedioso que es trabajar con ellas. Lo dejaremos en el directorio *./js/mitienda/util.js*. Exportamos ambas funciones.

```
function setCookie(cname, cvalue, exdays) {
  const d = new Date();
  d.setTime(d.getTime() + (exdays * 24 * 60 * 60 * 1000));
  const expires = `expires=${d.toUTCString()}`;
  document.cookie = `${cname}=${cvalue};${expires};path=/`;
}

function getCookie(cname) {
  const re = new
RegExp(`(?:(?:^|.*;\\s*)${cname}\\s*\\=\\s*([^;]*).*$)|^.*$`);
  return document.cookie.replace(re, '$1');
}

export { setCookie, getCookie };
```

2.2. Mostrar el mesaje

Debemos detectar si existe la cookie de aceptación del mensaje en el controlador del manager. Importamos en el contralador la función `getCookie()`.

```
import {getCookie} from '../util.js';
```

Lo debemos hacer una única vez al comienzo de la aplicación, por lo que lo haremos en evento `onLoad()`. Si no existe la cookie mostraremos el mensaje desde la vista.

```
if (getCookie('accetedCookieMessage') !== 'true') {
  this[VIEW].showCookiesMessage();
}
```

Comenzamos a modificar la vista importando la función `setCookie()`.

```
import {setCookie} from '../util.js';
```

Añadimos el método en la vista para generar la notificación.

```
showCookiesMessage() {
  const toast = `<div class="fixed-top p-5 mt-5">
    <div id="cookies-message" class="toast fade show bg-dark text-white
w-100 mw-100" role="alert" aria-live="assertive" aria-atomic="true">
      <div class="toast-header">
        <h4 class="me-auto">Aviso de uso de cookies</h4>
        <button type="button" class="btn-close" data-bs-dismiss="toast"
aria-label="Close" id="btnDismissCookie"></button>
      </div>
      <div class="toast-body p-4 d-flex flex-column">
        <p>
          Este sitio web almacenda datos en cookies para activar su
funcionalidad, entre las que se encuentra
          datos analíticos y personalización. Para poder utilizar este
sitio, estás automáticamente aceptando
          que
          utilizamos cookies.
        </p>
        <div class="ml-auto">
          <button type="button" class="btn btn-outline-light mr-3 deny"
id="btnDenyCookie" data-bs-dismiss="toast">
            Denegar
          </button>
          <button type="button" class="btn btn-primary"
id="btnAcceptCookie" data-bs-dismiss="toast">
            Aceptar
          </button>
        </div>
      </div>
    </div>
  </div>`;
  document.body.insertAdjacentHTML('afterbegin', toast);
}
```

En la notificación tenemos tres botones que al cliquearse la cerraran, para facilitar este proceso, Bootstrap dispone del atributo personalizado que `data-bs-dismiss="toast"` que permite hacerlo sin nuestra intervención.

Vamos a gestionar los eventos producidos en la notificación. Al cerrarse, independientemente del método, Bootstrap genera el evento `hidden.bs.toast`. Lo capturamos para eliminar la notificación del árbol DOM de la página.

```
const cookiesMessage = document.getElementById('cookies-message');
cookiesMessage.addEventListener('hidden.bs.toast', (event) => {
  event.currentTarget.parentElement.remove();
});
```

En los botones de cerrar y denegar añadimos un manejador para el evento `click` que elimine el contenido de la página y muestre un mensaje que es necesario aceptar el uso de las cookies.

```
const denyCookieFunction = (event) => {
  this.main.replaceChildren();
  this.main.insertAdjacentHTML('afterbegin', `<div class="container my-
3"><div class="alert alert-warning" role="alert">
      <strong>Para utilizar esta web es necesario aceptar el uso de
cookies. Debe recargar la página y aceptar las condicones para seguir
navegando. Gracias.</strong>
    </div></div>`);
  this.categories.remove();
  this.menu.remove();
};
const btnDenyCookie = document.getElementById('btnDenyCookie');
btnDenyCookie.addEventListener('click', denyCookieFunction);
const btnDismissCookie = document.getElementById('btnDismissCookie');
btnDismissCookie.addEventListener('click', denyCookieFunction);
```

Por último, para el botón de aceptación, capturamos el evento `click` para dejar la cookie que evite mostrar el mensaje nuevamente en la página.

```
const btnAcceptCookie = document.getElementById('btnAcceptCookie');
btnAcceptCookie.addEventListener('click', (event) => {
  setCookie('accetedCookieMessage', 'true', 1);
});
```

Añadimos unos selectores de CSS para dar formato a los botones.

```
button.deny {
    background-color: rgb(248, 249, 250);
    border-color: rgb(248, 249, 250);
    color: rgb(52, 58, 64)
}
button.deny:hover {
    background-color: rgb(52, 58, 64);
    border-color: rgb(248, 249, 250);
    color: rgb(248, 249, 250);
}
```

Lo preferible en una aplicación sería detectar esta cookie desde el servidor.

3. Servicio de autenticación

Vamos a simular un servicio de autenticación de usuarios que nos permita abrir y cerrar sesiones en nuestro proyecto. En base a si estamos autenticados o no, habilitaremos las funciones de administración del usuario.

3.1. Entidad User

Creamos una nueva entidad para la gestionar objetos de tipo usuario denominada `User`. Creamos el fichero *user.js* en el directorio de entidades.

```
import {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
} from '../exceptions.js';

class User {
  // Campos privados
  #username;
  #preferences;
  constructor(username) {
    if (!new.target) throw new InvalidAccessConstructorException();
    if (!username) throw new EmptyValueException('username');
    this.#username = username;

    Object.defineProperty(this, 'username', {
      enumerable: true,
      get() {
        return this.#username;
      },
    });
    Object.defineProperty(this, 'preferences', {
      enumerable: true,
      get() {
        return this.#preferences;
      },
      set(value) {
        if (!value) throw new EmptyValueException('preferences');
        this.#preferences = value;
      },
    });
  }
}

export { User };
```

3.2. Módulo de servicio de autenticación

Simulamos un servicio de autenticación en la ruta *mitienda/authentication/authentication.js*. Se trata de un patrón Singleton para validar un usuario denominado *admin* con contraseña *admin* y una factoría de objetos `User`.

```
import {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
} from '../exceptions.js';
import { User } from '../entities/user.js';

class AuthenticationServiceException extends BaseException {
  constructor(message = 'Error: Authentication Service Exception.',
fileName, lineNumber) {
    super(message, fileName, lineNumber);
    this.name = 'AuthenticationServiceException';
  }
}

const AuthenticationService = (function () {
  let instantiated;

  function init() { // Inicialización del Singleton
    class Authentication {
      constructor() {
        if (!new.target) throw new InvalidAccessConstructorException();
      }

      validateUser(username, password) {
        return !!((username === 'admin' && password === 'admin'));
      }

      getUser(username) {
        let user = null;
        if (username === 'admin') user = new User('admin');
        return user;
      }
    }

    const auth = new Authentication();
    Object.freeze(auth);
    return auth;
  }
```

```
  return {
    getInstance() {
      if (!instantiated) {
        instantiated = init();
      }
      return instantiated;
    },
  };
}());
export default AuthenticationService;
```

El controlador del manager necesita tener una instancia de este servicio. En *ManagerApp* importamos el servicio.

```
import AuthenticationService from '../authentication/authentication.js';
```

Al crear la instancia del controlador lo pasamos como argumento.

```
const ManagerApp = new ManagerController(Manager.getInstance(), new
ManagerView(), AuthenticationService.getInstance());
```

Cambiamos el controlador para aceptar el argumento en el constructor. Creamos dos `Symbol` para los campos privados del servicio y del usuario autenticado.

```
const AUTH = Symbol('AUTH');
const USER = Symbol('USER');
```

La firma del constructor cambia para tener el argumento con el servicio y hacer la asignación den campos privados.

```
constructor(model, view, auth) {
  this[MODEL] = model;
  this[VIEW] = view;
  this[AUTH] = auth;
  this[USER] = null;
  ...
}
```

3.3. Formulario de autenticación

Vamos a crear el formulario de autenticación.

3.3.1. Enlace de identificación

Tenemos que crear un enlace para abrir el formulario de autenticación. Comenzamos añadiendo una serie de nuevos selectores de CSS en *main.css*.

```
#login {
    cursor: pointer;
    display: block;
    font-family: "Open Sans", sans-serif;
    font-size: 1.1rem;
    font-weight: 400;
    padding: 8px;
    height: 150px;
}
```

```
.user_card {
    width: 350px;
    margin-top: 20px;
    margin-bottom: 20px;
    background:  #faa541;
    position: relative;
    display: flex;
    justify-content: center;
    flex-direction: column;
    padding: 10px;
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
    -webkit-box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
    -moz-box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
    border-radius: 5px;
}
.brand_logo_container {
    position: absolute;
    height: 170px;
    width: 170px;
    top: -75px;
    border-radius: 50%;
    background: #60a3bc;
    padding: 10px;
    text-align: center;
}
.brand_logo {
    height: 150px;
    width: 150px;
    border-radius: 50%;
    border: 2px solid white;
}
.form_container {
    margin-top: 20px;
}
.login_btn {
    width: 100%;
    background: #c0392b !important;
    color: white !important;
}
.login_btn:focus {
    box-shadow: none !important;
    outline: 0px !important;
}
```

```
.login_container {
    padding: 0 2rem;
}
.input-group-text {
    background: #c0392b !important;
    color: white !important;
    border: 0 !important;
    border-radius: 0.25rem 0 0 0.25rem !important;
}
.input_user,
.input_pass:focus {
    box-shadow: none !important;
    outline: 0px !important;
}
.custom-checkbox .custom-control-input:checked~.custom-control-
label::before {
    background-color: #c0392b !important;
}
```

En *index.html* eliminamos el contenido de la capa donde mostramos la imagen de usuario y le asignamos un identificador.

```
<div id="userArea" class="my-1 col-lg-2 header-nav-utilities d-inline-
flex align-items-center justify-content-lg-end"></div>
```

Creamos una vista para el link de login y un *bind* para dotarle de funcionalidad.

```
showIdentificationLink() {
  const userArea = document.getElementById('userArea');
  userArea.replaceChildren();
  userArea.insertAdjacentHTML('afterbegin', `<div class="account d-flex
mx-2 flex-column" style="text-align: right; height: 40px">
    <a id="login" href="#"><i class="bi bi-person-circle" aria-
hidden="true"></i> Identificate</a>
  </div>`);
}

bindIdentificationLink(handler) {
  const login = document.getElementById('login');
  login.addEventListener('click', (event) => {
    this[EXCECUTE_HANDLER](handler, [], 'main', { action: 'login' }, '#',
event);
  });
}
```

Ambos métodos los debemos invocar desde el `onLoad()`. Aunque no disponemos de cookie de usuario todavía, dejamos el código preparado para chequearla.

```
if (getCookie('activeUser')) {

} else {
  this[VIEW].showIdentificationLink();
  this[VIEW].bindIdentificationLink(this.handleLoginForm);
}
```

En el método bind le asignamos el manejador que permita crear la vista del formulario que realizaremos en el siguiente punto.

3.3.2. Creación del formulario de login

Creamos un manejador en el controlador que nos permita abrir el formulario. Invocaremos al bind para dotarle de funcionalidad más adelante.

```
handleLoginForm = () => {
  this[VIEW].showLogin();
  //this[VIEW].bindLogin(this.handleLogin);
};
```

Creamos la vista del formulario que estamos invocando.

```
showLogin() {
  this.main.replaceChildren();
  const login = `<div class="container h-100">
    <div class="d-flex justify-content-center h-100">
      <div class="user_card">
        <div class="d-flex justify-content-center form_container">
        <form name="fLogin" role="form" novalidate>
            <div class="input-group mb-3">
              <div class="input-group-append">
                <span class="input-group-text"><i class="bi bi-person-
circle"></i></span>
              </div>
              <input type="text" name="username" class="form-control
input_user" value="" placeholder="usuario">
            </div>
            <div class="input-group mb-2">
              <div class="input-group-append">
                <span class="input-group-text"><i class="bi bi-key-
fill"></i></span>
              </div>
              <input type="password" name="password" class="form-control
input_pass" value="" placeholder="contraseña">
            </div>
            <div class="form-group">
              <div class="custom-control custom-checkbox">
                <input name="remember" type="checkbox" class="custom-
control-input" id="customControlInline">
```

```
                <label class="custom-control-label"
for="customControlInline">Recuerdame</label>
              </div>
            </div>
              <div class="d-flex justify-content-center mt-3
login_container">
                <button class="btn login_btn"
type="submit">Acceder</button>
          </div>
          </form>
        </div>
      </div>
    </div>
  </div>`;
  this.main.insertAdjacentHTML('afterbegin', login);
}
```

Integramos en el `history` la navegación en el formulario añadiendo la acción en el array.

```
login: () => ManagerApp.handleLoginForm()
```

3.3.3. Autenticación

Vamos a dotar de funcionalidad al formulario. Tenemos que capturar el evento submit del formulario para pasar el nombre de usuario y la contraseña la manejador que realice la validación. Implementamos el bind que realiza esta función.

```
bindLogin(handler) {
  const form = document.forms.fLogin;
  form.addEventListener('submit', (event) => {
    handler(form.username.value, form.password.value);
    event.preventDefault();
  });
}
```

En el método `handleLoginForm()` del controlador descomentamos la invocación del bind.

```
this[VIEW].bindLogin(this.handleLogin);
```

Creamos el manejador que realice la validación. Si el usuario y la contraseña son correctos, recuperaremos un objeto `User` del servicio de autenticación y lanzaremos un evento de que la sesión se ha creado con `onOpenSession()`, de momento no lo tenemos creado y permanece comentado.

```
handleLogin = (username, password) => {
  if (this[AUTH].validateUser(username, password)) {
    this[USER] = this[AUTH].getUser(username);
    // this.onOpenSession();
  } else {
    this[VIEW].showInvalidUserMessage();
  }
};
```

En caso de no ser válidos el nombre de usuario y la contraseña debemos mostrar un mensaje indicando la circunstancia. Creamos esta vista.

```
showInvalidUserMessage() {
  this.main.insertAdjacentHTML('beforeend', `<div class="container my-
3"><div class="alert alert-warning" role="alert">
  <strong>El usuario y la contraseña no son válidos. Inténtelo
nuevamente.</strong>
</div></div>`);
  document.forms.fLogin.reset();
  document.forms.fLogin.username.focus();
}
```

Implementamos el evento `onOpenSession()`. Vamos a ir añadiéndole funcionalidad, pero previamente no debemos olvidar descomentar su invocación en el manejador `handleLogin()`. En primer lugar, tiene que reiniciar el manager y shoppingcart.

```
onOpenSession(){
  this.onInit();
  ShoppingCartApp.onInit();
}
```

Tenemos que reemplazar en el `history` la acción de inicio. Creamos un método en la vista que lo realice.

```
initHistory() {
  history.replaceState({ action: 'init' }, null);
}
```

Y lo invocamos desde `onOpenSession()`.

```
this[VIEW].initHistory();
```

Mostramos el perfil del usuario. Se trata de una vista que recibe como argumento el objeto `User` que tenemos almacenado en el controlador.

```
showAuthUserProfile(user) {
  const userArea = document.getElementById('userArea');
  userArea.replaceChildren();
  userArea.insertAdjacentHTML('afterbegin', `<div class="account d-flex
mx-2 flex-column" style="text-align: right">
      ${user.username} <a id="aCloseSession" href="#">Cerrar sesión</a>
    </div>
    <div class="image">
      <img alt="${user.username}" src="img/user.jpg" />
    </div>`);
}
```

La invocamos en `onOpenSession()`.

```
this[VIEW].showAuthUserProfile(this[USER]);
```

Por último, los menús de administración solo deben aparecer si el usuario está autenticado. Movemos los siguientes métodos desde `onLoad()` a `onOpenSession()`.

```
this[VIEW].showAdminMenu();
this[VIEW].bindAdminMenu(
  this.handleNewCategoryForm,
  this.handleRemoveCategoryForm,
  this.handleNewProductForm,
  this.handleRemoveProductForm,
  this.handleRemoveProductForm2,
);
```

3.4. Gestión de cookie del usuario

Vamos a habilitar una cookie de usuario para autenticarlo de manera directa al abrir la aplicación. Si recordamos, la cookie la hemos denominado `activeUser`. En la vista creamos un método que deje esta cookie, el valor será el nombre del usuario. En una aplicación real utilizaríamos un identificador de sesión.

```
setUserCookie(user) {
  setCookie('activeUser', user.username, 1);
}
```

Siguiendo en la vista, el bind `bindLogin()` invoca el manejador `handleLogin()` que realiza la validación del usuario, le pasamos el valor recogido del *input check* del formulario.

```
handler(form.username.value, form.password.value, form.remember.checked);
```

Cambiamos la firma del manejador `handleLogin()`.

```
handleLogin = (username, password, remember) => {
```

En el caso de tener credenciales válidas, dejaremos la cookie.

```
if (remember) {
  this[VIEW].setUserCookie(this[USER]);
}
```

En el evento `onload()`, modificamos el condicional de la gestión de la cookie de usuario para que al detectar la cookie lance el evento `onOpenSession()`.

```
const userCookie = getCookie('activeUser');
if (userCookie){
  const user = this[AUTH].getUser(userCookie);
  if (user){
    this[USER] = user;
    this.onOpenSession();
  }
} else {
  this[VIEW].showIdentificationLink();
  this[VIEW].bindIdentificationLink(this.handleLoginForm);
}
```

3.5. Cerrar la session

La acción de cerrar la sesión implica el borrado de la cookie `activeUser` además de crear un evento que realice todas las acciones para restaurar el estado inicial de la aplicación.

En la vista creamos el método para borrar la cookie.

```
deleteUserCookie(){
  setCookie('activeUser','',0);
}
```

Seguimos con la vista. Creamos un método que borre el enlace de administración.

```
removeAdminMenu() {
  const adminMenu = document.getElementById('adminMenu');
  if (adminMenu) adminMenu.parentElement.remove();
}
```

Creamos un bind para dotar de funcionalidad al link de cierre de sesión. El manejador ejecutado se encargar de realizar la acción.

```
bindCloseSession(handler){
  document.getElementById('aCloseSession').addEventListener('click',
(event) => {
    handler();
    event.preventDefault();
  });
}
```

Este bind los invocamos desde onOpenSession() después de mostrar el perfil del usuario.

```
this[VIEW].bindCloseSession(this.handleCloseSession);
```

El manejador debe invocar el evento de cierre de sesión.

```
handleCloseSession = () => {
  this.onCloseSession();
  this.onInit();
  ShoppingCartApp.onInit();
  this[VIEW].initHistory();
};
```

El evento de cierre de sesión libera el usuario del controlador, borra la cookie de usuario y regenera la cabecera.

```
onCloseSession() {
  this[USER] = null;
  this[VIEW].deleteUserCookie();
  this[VIEW].showIdentificationLink();
  this[VIEW].bindIdentificationLink(this.handleLoginForm);
  this[VIEW].removeAdminMenu();
}
```

Por último, en el `onLoad()` invocamos el cierre de sesión en caso de no tener cookie ya que las acciones que estábamos invocando se realizan en el cierre.

```
this.onCloseSession();
```

3.6. Gestión de la seguridad

Vamos a evitar que las acciones que necesiten privilegios puedan ser ejecutadas sin que el usuario este autenticado. Comenzamos creando una vista que informe del error.

```
showErrorUserMessage() {
  this.main.replaceChildren();
  this.main.insertAdjacentHTML('beforeend', `<div id="error"
class="container my-3"><div class="alert alert-warning" role="alert">
    <strong>Para realizar esta acción es necesario estar
autenticado.</strong>
  </div></div>`);
  document.getElementById('error').scrollIntoView();
}
```

Ahora debemos modificar cada uno de los manejadores que necesitan privilegios para comprobar que tenemos un usuario autenticado, si no es así mostraremos el error. Modificamos los manejadores de la siguiente forma para que el cuerpo del manejador esté en la cláusula `else`:

```
if (!this[USER]) {
  this[VIEW].showErrorUserMessage();
} else {
  ...
}
```

Modificamos los siguientes manejadores:

- `handleNewCategoryForm()`
- `handleCreateCategory()`
- `handleRemoveCategoryForm()`
- `handleRemoveCategory()`
- `handleNewProductForm()`
- `handleCreateProduct()`
- `handleRemoveProductForm()`
- `handleRemoveProductForm2()`
- `handleRemoveProductListByType()`
- `handleRemoveProductListByCategory()`
- `handleRemoveProductListByTypeCategory()`
- `handleRemoveProduct()`

4. Idioma de la página

Vamos a elegir el idioma de la página y que se recuerde la próxima vez que abramos la aplicación, para ello vamos a utilizar el api **WebStorage**. El cambio solo va a implicar mostrar la bandera del idioma seleccionado. En la cabecera disponemos con un desplegable con varias banderas para elegir el idioma. La aplicación debe recordar qué idioma estamos utilizando.

En el controlador del manager creamos un nuevo campo privado para el idioma.

```
const LANGUAGE = Symbol('LANGUAGE');
```

En el constructor hacemos la asignación para que valor por defecto sea español.

```
this[LANGUAGE] = 'es';
```

Creamos un bind para dotar de funcionalidad al desplegable de las banderas. Invocaremos el manejador para cambiar el idioma aprovechando el atributo personalizado. Además, dejaremos en `localStorage` el idioma elegido para su lectura en la carga de la aplicación.

```
bindLanguageSelection(handler) {
  const lFlags =
document.getElementById('language').querySelectorAll('ul.dropdown-menu
a');
  for (const link of lFlags) {
    link.addEventListener('click', (event) => {
      const { language } = event.currentTarget.dataset;
      handler(language);
      localStorage.setItem('language', language);
      event.preventDefault();
    });
  }
}
```

Invocamos el bind en el `onLoad()`.

```
this[VIEW].bindLanguageSelection(this.handleChangeLanguage);
```

El manejador handleChangeLanguage() será el encargado de realizar el cambio de idioma, actualizando el campo privado y lanzando un evento de cambio de idioma.

```
handleChangeLanguage = (language) => {
  this[LANGUAGE] = language;
  this.onChangeLanguage();
};
```

El evento `onChangeLanguage()` debería realizar todas las acciones para cambiar el idioma. Para nuestro ejemplo tan solo actualizamos la bandera.

```
onChangeLanguage() {
  this[VIEW].showSelectedLanguage(this[LANGUAGE]);
}
```

Creamos la vista para actualizar la bandera.

```
showSelectedLanguage(language) {
  const selectedLanguage = document.querySelector('#languageDropdown
img');
  selectedLanguage.src = `img/languages/${language}.png`;
}
```

Para recuperar el idioma seleccionado previamente creamos un método en la vista.

```
getLanguage() {
  return localStorage.getItem('language');
}
```

Por último, detectamos el idioma en el evento de carga.

```
const language = this[VIEW].getLanguage();
if (language) {
  this[LANGUAGE] = language;
  this.onChangeLanguage();
}
```