Promesas:

-son pedidos asincrónicos. Pueden hacer dos acciones (hay una act. paralela a la principal). Hay otros pedidos asincrónicos. Ej: Fetch
-la función asincrónica puede o no devolver un resultado. Pero el código que le sigue a la función asincrónica se sigue ejecutando

El .then es obligatorio en una función asincrónica. Captura el resultado (info). Adentro del .then es donde vamos a poder procesar lo que devuelva el findByPk (que es una promesa). Lo que va adentro es un parámetro (data), lo podemos llamar como queramos. Ahí lo que hacemos es pasarle la info. a la vista (ej: productEdit) a través de un objeto literal. Product (lo elegimos nosotos): data (coincide con el parámetro).
Es decir: estamos yendo a buscar la info(data) a través de findByPk a la base de datos del modelo product

```
edit: function(req, res){
        //control de acceso

        if(req.session.user ==undefined){
            return res.redirect("/login");
        } else{
            db.Product.findByPk(req.params.id)
            .then( data => {
                return res.render('productEdit', {product:data});
            })
            .catch(error => {
                console.log(error);
            }

            )
        }
```

A veces los .then() pueden tener otras promesas adentro. Se usa otro .then que se ejecuta cuando ya lo haya hecho el otro. Los .then encadenados necesitan retornar la data procesada para que pueda ser usada por el siguiente .then()

El .catch captura cualquier error. Dentro del método decidimos que hacer con el error. Con console.log se muestra en la consola

promise.all procesa muchas promesas al mismo tiempo (no creo que la usemos)

Hay otros lenguajes que son sincrónicos y no permiten una act. paralela

Sequelize: es un ORM (copia la estructura de las tablas) y nos ayuda a conectarnos e interactuar con la base de datos, como MySQL.

Como es un paquete utilizado por NodeJS se utiliza NPM

Tiene 4 pasos que hay que poner en la terminal (chequear en slides)
Después hay que modificar el archivo config.js para que sea un módulo exportable

Hay que requerir /database/models cada vez que utilicemos sequelize para realizar consultas a la base de datos. Requerimos la conexión a la base de datos y los modelos creados

```
const db = require('../database/models');
```

Modelos:

```
db.Product.findByPk
```

A la variable db le decimos que queremos trabajar con uno de los modelos. Ponemos el alias="Product" y trabajamos con funciones (findAll, etc). Esto es una promesa. Todos los métodos que ejecutamos de sequelize son promesas. Como ya dije, después usamos el .then para procesar lo que devuelva el .findAll

Los modelos:
-Contienen info. pura de configuración y acceden a la capa de almacenamiento de datos.
-Creamos un modelo para cada tabla de nuestra base de datos. Con mayúscula.
-Un modelo es una función que tenemos que definir y exportar con module.exports
-Recibe 2 parámetros

-sequelize: para acceder a su método define()

```
const Comment = sequelize.define(alias, cols, config);
```

-dataTypes: nos permite el tipo de dato que contienen las columnas de las tablas

```
type: dataTypes.INTEGER
```

-El método define() nos permite definir asignaciones entre un modelo y una tabla. Recibe 3 parámetros

-Primero hay que definir un alias, con este alias, sequelize va a identificar al modelo
let alias = "Comment" (nombre del archivo)
-El segundo parámetro, cols, tiene la definición de los tipos de datos que recibe la columna: STRING: texto, INTEGER: número

-Después hay una configuración adicional necesaria para que sequelize funcione correctamente. Dentro de un objeto literal. Por ej: si la tabla no tiene los campos de timestamps (timestamps: false)

let config = { etc etc

LOS TIMESTAMPS (createdAt, updatedAt) son opcionales. Son campos que guardan la fecha de creación y última edición.

-Almacenamos el retorno de define() en la variable con el nombre del modelo
const Comment = sequelize.define (alias, cols, config)

-Y después hay que retornar para que funcione
return Comment;

Luego de todo esto hay que ir a un controlador

Find

-Sequelize utiliza una función llamada FIND para buscar info. en la base de datos:
-findAll()
-findByPk()
-findOne()

db.Product.findAll() //findAll me trae todos los registros de una tabla, devuelve una promesa, por eso usamos .then()
La resolución de la promesa se pone en el parámetro del callback.
-puede recibir criterios de búsqueda adentro de los paréntesis

```
search: function (req, res){
        let data = req.query.search;

        db.Product.findAll({

            include: [{association: "user"},
            {association: "comments"}],

            where: [
                {productName: {[op.like]: '%'+data+'%'}}

        ]})
```

```
        .then(data => {
            return res.render('search-results', {product: data});
        }
```

findByPk(): busca un registro que coincida con un valor de una clave primaria de la tabla. Es el equivalente a escribir la sentencia sql: SELECT * FROM movies WHERE id= 42

```
else{

    db.User.findByPk(userId)
    .then (function(user){
        return res.render('userEdit' , {userEdit: user})
    })
    .catch(e => {console.log(e)})
```

findOne(): obtiene un registro que coincida con algún criterio de búsqueda. Tenemos que pasar los criterios como un objeto literal dentro del método. Dentro de where ponemos para buscar el email

```
login: function(req, res){

        // Buscar el usuario que se quiere loguear.
        db.User.findOne({
            where: [{email: req.body.email}]
        })
        .then( user => {
            let errors = {};
            //¿Está el email en la base de datos
            if(user == null){
                //crear un mensaje de error
                 errors.message = 'El email no existe'
                //pasar mensaje a la vista
                 res.locals.errors = errors
                //renderizar esa vista
                 return res.render('login');
            } else if(bcrypt.compareSync(req.body.password,
user.password) == false){
            //crear un mensaje de error
            errors.message = 'La contraseña es incorrecta'
            //Pasar el mensaje a la vista
            res.locals.errors = errors
            //renderizar la vista
            return res.render('login');
```

Sequelize WHERE.

-utilizamos WHERE (objeto literal) para filtrar datos.

-Hay un objeto literal y adentro puede tener:
  -el atributo WHERE que va adentro de findAll o findOne
  -dentro del WHERE pasamos un array con un objeto literal, su atributo es la columna de la tabla y el valor es el valor a buscar

Esto filtra los datos que coincidan con el valor buscado. Filtra por “igualdad”.
Para filtrar por criterios que no sean “igualdad” necesitamos utilizar los operadores de sequelize

```
const op = db.Sequelize.Op;
```

Después usamos operadores en cada filtro del WHERE

```
search: function (req, res){
        let data = req.query.search;

        db.Product.findAll({

            include: [{association: "user"},
            {association: "comments"}],

            where: [
                {productName: {[op.like]: '%'+data+'%'}}

        ]})
        .then(data => {
            return res.render('search-results', {product: data});
        })
        .catch(error => {
            console.log(error);
        })
```

{[op.like]: ‘&’+data+’&”}} ponemos alguna palabra que esté en el nombre del producto y la obtenemos

También hay otros operadores: gte, or

Sequelize: order, limit y offset

El atributo order: organiza de manera descendente o ascendente los registros que se obtienen cuando haces una consulta a la base de datos

```
order: [['comments', 'id', 'desc']]
```

El primer valor es la columna sobre la que se desea ordenar, el segundo es el id, el tercero es el criterio de ordenamiento (descendente)

El atributo limit: sirve para limitar el número de resultados a obtener. Puede ir adentro de findAll() o findOne(): (no lo usamos)

El atributo offset: sirve para omitir resultados. Se usa en findAll() o en findOne()

Se pueden usar todas las condiciones juntas

contiene parámetros de ruta

```
req.params.id
```

El productName viene siempre del formulario. req.body para traer del formulario

```
productName: req.body.productName
```

req.query contiene los parámetros de consulta de URL

req.session

res.redirect()	Redirecciona a la vista.
res.render()	Renderiza a la vista.
res.send()	Envía una respuesta de varios tipos.