

La interfaz de usuario (*UI*), en el campo del diseño industrial de la interacción human computadora, es el espacio donde ocurren las interacciones entre humanos y máquinas. En definitiva, La *UI* es una serie de pantallas, páginas y elementos visuales —como botones e íconos— que son utilizados para interactuar con un dispositivo.

La Experiencia del Usuario (*UX*), por otro lado, es la experiencia interna que una persona tiene mientras interactúa con todos los aspectos de un producto o servicio (alta subjetividad).

&& :: || :: DESKTOP :: || :: &&

Componentes:

- *TextBox* - Input de texto
- *Label* - Etiqueta de texto
- *Button*
- *Panels* - Contenedor de componentes
- *Windows y Dialogs*
- *Error Panel*

-**Window**: Es la clase abstracta para todas las ventanas.

-**MainWindow**: Es un tipo especial de ventana que se usa para aplicaciones simples o de una sola ventana.

-**SimpleWindow**: Ventana común que agrega el panel de errores.

-**Dialog**: Es una ventana final que depende de alguna de las anteriores y que debe generar una acción y cerrarse.

```
class LoginWindow : MainWindow<Login>(Login()) {
    override fun createContents(panel: Panel) {
        this.title = "Login del sistema"
        Label(panel).setText("Ingresa el usuario")
        TextBox(panel)
        Label(panel).setText("Ingresa el password")
        PasswordField(panel)
        Button(panel).setCaption("Autenticar")
    }
}

fun main() {
    LoginWindow().startApplication()
}
```

Pero las ventanas *Window* no pueden ser inicializadas desde un *main()* como sí sucede con *MainWindow*. Es necesario otra estrategia de inicialización, para lo que hay que usar la clase *Application* que se encarga de inicializar la aplicación y llamar a la *Window* que indiquemos como “inicial”. Luego vamos a poder interactuar entre *Windows* y *Dialogs* libremente.

Application + SimpleWindows

```
class Main : Application() {  
    override fun createMainWindow(): MainWindow {  
        return MainWindow(this, User())  
    }  
}  
  
fun main() { Main().start() }  
  
class MainWindow : SimpleWindow<User>{  
    constructor (owner: WindowOwner, model: User ) : super(owner, model)  
    override fun addActions(actionsPanel : Panel) { /* Actions Panel */ }  
    override fun createFormPanel(mainPanel : Panel) {  
        Label(mainPanel).setText("Prueba SimpleWindow")  
    }  
}
```

Application + Dialog

```
class FormWindow : SimpleWindow<User> {  
    constructor (owner: WindowOwner, model: User) : super(owner, model)  
    override fun createFormPanel(mainPanel: Panel) {  
        Button(mainPanel).onClick(Action {  
            AddNameDialog(this, this.modelObject).open()  
        })  
    }  
}  
  
class AddNameDialog : Dialog<User> {  
    constructor(owner: WindowOwner, model: User ) : super(owner, model)  
    override fun createFormPanel(panel : Panel ) {  
        Label(panel).setText("Dialog Example: ")  
        Button(panel).setCaption("Aceptar").onClick(Action { this.close() })  
    }  
}
```

Eventos:

- Cualquier suceso en el sistema
- Comunicación entre dos componentes
- Inversión de la relación de conocimiento
- Tanto la vista como el modelo, pueden producirlos
- Similar patrón *Observer*.

Binding

- Mecanismo que permite vincular o enlazar propiedades del modelo con la UI.
- Relaciona dos propiedades manteniendolas sincronizadas
- El binding puede ser direccional o bidireccional
- Muchos *framework* proveen mecanismos de *binding*.

En la UI podemos “bindear” una propiedad con el modelo de la siguiente forma:

```
Label(panel).bindValueToProperty<String, ControlBuilder>("user")
```

//En el modelo con el annotation

```
@Observable
```

```
class Login {
```

```
    var user: String = ""
```

```
}
```

```
@Observable
```

```
class Login {
```

```
    var user: String = ""
```

```
    set(value) { resetAuth(); field = value }
```

```
    var password: String = ""
```

```
    set(value) { resetAuth(); field = value }
```

```
    var authenticated: Boolean = false
```

```
    var authenticatedColor: Color = Color.ORANGE
```

```
    fun authenticate() {
```

```
        authenticated = (user == "polo") && (password == "polo")
```

```
    authenticatedColor =
```

```
        if (authenticated) Color.GREEN else Color.ORANGE
```

```
}
```

```
    private fun resetAuth() {
```

```
        authenticated = false
```

```
        authenticatedColor = Color.ORANGE
```

```
}
```

```
class LoginWindow : MainWindow<Login>(Login()) {
```

```
    override fun createContents(panel: Panel) {
```

```
        this.title = "Login del sistema"
```

```
        Label(panel).setText("Ingrese el usuario")
```

```
        TextBox(panel).bindValueToProperty<Int, ControlBuilder>("user")
```

```
        Label(panel).setText("Ingrese el password")
```

```
        PasswordField(panel).bindValueToProperty<Int, ControlBuilder>("password")
```

```
        Button(panel)
```

```
            .setCaption("Autenticar")
```

```
            .onClick { modelObject.authenticate() }
```

```
        val status = Label(panel)
```

```
        status.bindValueToProperty<Double, ControlBuilder>("authenticated")
```

```
        status.bindBackgroundToProperty<ControlBuilder, Any,
```

```
Any>("authenticatedColor")
```

```
    }
```

```
}
```

```
fun main() {
```

```
    LoginWindow().startApplication()
```

```
}
```

Patrón Model-View-Controller

Patrón de diseño o modelo de abstracción utilizado para definir y estructurar los componentes necesarios en el desarrollo de software.

Vista

- ▷ Capa de presentación / interfaz de usuario.
- ▷ Ventanas, botones, *inputs*.

Modelo

- ▷ Capa que contiene la lógica de la aplicación.
- ▷ Contiene la lógica de negocio/reglas de negocio.
- ▷ Modelo de objetos.

Controlador

- ▷ Intermediario entre la capas Vista y Modelo.
- ▷ Orquestadores del modelo o decidores de la operación
- ▷ Gestiona el flujo de información entre el las capas.
- ▷ Adapta la vista con el modelo.

```
class LoginWindow : MainWindow<Login>(Login()) {  
    override fun createContents(panel: Panel) {  
        Label(panel).setText("Ingresa el usuario")  
        TextBox(panel).bindValueToProperty<Int, ControlBuilder>("user")  
        Label(panel).setText("Ingresa el password")  
        PasswordField(panel).bindValueToProperty<Int, ControlBuilder>("password")  
        Button(panel)  
            .setCaption("Autenticar")  
            .onClick { modelObject.authenticate() }  
        val status = Label(panel)  
        status.bindValueToProperty<Double, ControlBuilder>("authenticated")  
        status.bindBackgroundToProperty<ControlBuilder, Any,  
Any>("authenticatedColor")  
    }  
}
```

Application Model vs Model Simple - Ejemplo

@Observable

```
class FilmCreateAppModel{  
    var film : Film = Film()  
    var genders : MutableList<Gender> = ArrayList<Gender>()  
}
```

```
class Window : SimpleWindow<FilmCreateAppModel> {  
    constructor (owner: WindowOwner, model: FilmCreateAppModel) : super(owner, model)  
    override fun createFormPanel(mainPanel: Panel) {  
        Label(mainPanel).setText("Nombre de la película")  
        TextBox(mainPanel).bindValueToProperty<String, ControlBuilder>("film")  
        Label(mainPanel).setText("Genero")  
    }  
}
```

```

        List<FilmCreateAppModel>(mainPanel).bindItemsToProperty("genders")
    }
}

```

Panel

- Es un contenedor de *widgets* (es una pequeña aplicación o programa, usualmente presentado en archivos o ficheros pequeños. Entre sus objetivos están dar fácil acceso a funciones frecuentemente usadas y proveer de información visual).
- Cada panel tiene:
 - Un modelo. Por defecto usa el modelo de vista.
 - Un *layout*. Por defecto, tiene un diseño vertical.

Layout (En español: Diseño o Disposición)

- Define cómo se organizan los controles a medida que se crean.
- Define cómo se van a acomodar los componentes visuales.
- No es un componente visual (no se lo puede “ver” directamente)

Pueden ser:

- **VerticalLayout**
 - Los componentes se disponen verticalmente, uno debajo del otro.
 - Es la disposición por defecto.
 - Muy útil para:
 - Ventanas simples
 - Paneles de lado



- **HorizontalLayout**

- Los componentes se crean uno al lado del otro, de izquierda a derecha.
- Muy útil para:
 - Botoneras
 - Columnas de tamaño variable

```

mainPanel.setLayout(HorizontalLayout())
Button(mainPanel).setCaption("Agregar")
Button(mainPanel).setCaption("Guardar")
Button(mainPanel).setCaption("Cancelar")

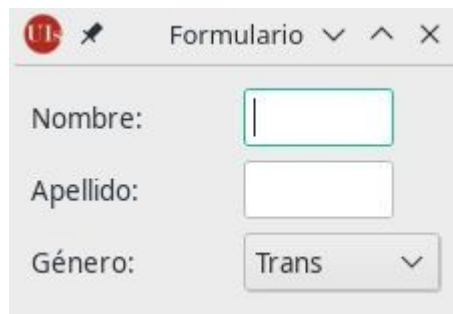
```



- ColumnLayout

- Agrupa los componentes en columnas.
- Se debe indicar la cantidad de columnas.
- Los componentes se crean uno al lado del otro (de izq a der) hasta completar la fila. Luego pasan a la siguiente fila.
- Muy útil para:
 - o Formularios
 - o Columnas de igual tamaño

```
Label(mainPanel).setText("Nombre:"); TextBox(mainPanel)
Label(mainPanel).setText("Apellido:"); TextBox(mainPanel)
Label(mainPanel).setText("Género:")
val selector = Selector<String>(mainPanel)
selector.bindValueToProperty<String, ControlBuilder>("genre")
selector.bindItems<String>(ObservableProperty(model, "genresList"))
```



Ventana Principal

Hasta el momento venimos usando una sola ventana:

- ▶ Creamos una `LoQueSeaWindow` que extienda de `MainWindow<T>`.
- ▶ Definimos el constructor pasándole el modelo.
- ▶ Sobreescribimos el método `createContents` con el contenido.
- ▶ Levantamos la aplicación desde el `main`.

```
class MainCompanyWindow : MainWindow<Company> {
    constructor(model: Company) : super(model)
    override fun createContents(mainPanel: Panel) {
        title = "Empresa"
        Label(mainPanel).setText("Empleados: ")
        val employees = List<Employee>(mainPanel)
        employees.bindValueToProperty<Employee, ControlBuilder>("empl")
        employees.bindItemsToProperty("employees")
    }
}

@Observable class Company {
    var empl: Employee? = null
    var employees = listOf(Employee("Jon Snow"), ...)
}
```

```
@Observable class Employee(private val name: String) {
    override fun toString() = name
}
```

Para levantar otra ventana:

- A partir de algún evento podemos instanciar una nueva ventana.
- Esta ventana debe conocer a la ventana que la invocó (solemos llamarla *owner* o *parent*).
- También debe tener un **modelObject** que puede ser “pasado” desde el *parent*.

Window ⇒ Dialog

```
class MainCompanyWindow : MainWindow<Company> {
    override fun createContents(mainPanel: Panel) {
        // Abro Dialog
        Button(mainPanel)
            .setCaption("Agregar White Walker")
            .onClick { edit() }
    }
    private fun edit() {
        val dialog = EmployeeDialog(this, modelObject.employees[0])
        dialog.onAccept {
            modelObject.employees.add(Employee("White Walker"))
        }
        dialog.onCancel { /* Do Nothing */ }
        dialog.open()
    }
}

class EmployeeDialog : Dialog<Employee> {
    constructor(owner WindowOwner, model: Employee) : super(owner, model)
    override fun addActions(actions: Panel) {
        Button(actions)
            .setCaption("Aceptar")
            .onClick { accept() }
        Button(actions)
            .setCaption("Cancelar")
            .onClick { cancel() }
    }
    override fun createFormPanel(mainPanel: Panel) {
        Label(mainPanel).setText(this.modelObject.name)
    }
}
```

Las ventanas *Dialog* funcionan como “modales”, o sea que se usan para una función específica y se cierran. Se pueden seguir abriendo ventanas o *dialogs* desde un *Dialog* pero no es recomendable porque se van “stackeando”. Para poder trabajar con ventanas independientes deben ser *Window* o *SimpleWindow*. Una *MainWindow* puede abrir una *Window* pero luego nunca más se puede volver a la *MainWindow*.

Windows => Windows

```
class CompanyWindow : SimpleWindow<Company> {
    constructor(parent: WindowOwner, model: Company) {
        super(parent, model)
    }
    override fun addActions(actionsPanel: Panel) {}
    override fun createFormPanel(mainPanel: Panel) {}
    override fun createContents(mainPanel: Panel) {
        title = "Empresa"
        Button(mainPanel)
            .setCaption("Abrir Empleado")
            .onClick {
                close()
                EmployeeWindow(this, modelObject.employees[0]).open()
            }
    }
}

class EmployeeWindow : SimpleWindow<Employee> {
    constructor(owner: WindowOwner, model: Employee) {
        super(owner, model)
    }
    override fun addActions(actionsPanel: Panel) {}
    override fun createFormPanel(mainPanel: Panel) {}
    override fun createContents(mainPanel: Panel) {
        title = "Empleado"
        Button(mainPanel)
            .setCaption("Volver a Empresa")
            .onClick {
                close()
                CompanyWindow(this, Company()).open()
            }
    }
}
```

Validaciones

Las validaciones son técnicas que permiten asegurar que los valores con los que se vaya a operar estén dentro de determinado dominio. Estas pueden ser:

- Validaciones a nivel de Vista (View)

- ▷ Aquellas que deben ser realizadas sin la necesidad de intervención del modelo
- ▷ La validación se realiza en la misma capa.
- ▷ Menos costosa.

- Validaciones a nivel de Modelo

- ▷ Aquellas que deben ser realizadas por el modelo dado que implican reglas de negocio.
- ▷ Deben ser informadas a la capa *View*.
- ▷ Mas costosa.
- ▷ Mensajes descriptivos que permitan al usuario identificar claramente cuál regla de negocio no se está cumpliendo y le permita continuar con el proceso.

En Arena las ventanas del tipo *SimpleWindows* manejan *UserException* mostrando el mensaje de error.

Excepciones

Manejo de errores

Son errores propios de la programación, que hace que no sean modelados, simplemente ocurren y que identificarlas y tratarlas

Problemas de Binding

Al querer bindear el *input* de la vista con el atributo del modelo podemos llegar a tener problemas

- De *Model* a *View* a veces funciona por el *toString()*
- De *View* a *Model* explota por tipado

Se resuelve utilizando un objeto que transforme:

- El objeto de modelo en *String*.
- El *String* en objeto de modelo O sea, un *Transformer*.

```
class DateTransformer : ValueTransformer<LocalDate, String> {
    private val pattern = "dd/MM/yyyy"
    private val formatter = DateTimeFormatter.ofPattern(pattern)
    override fun getModelType() = LocalDate::class.java
    override fun getViewType() = String::class.java
    override fun modelToView(valueFromModel: LocalDate): String =
        valueFromModel.format(formatter)
    override fun viewToModel(valueFromView: String): LocalDate {
        try {
            return LocalDate.parse(valueFromView, formatter)
        } catch (e: DateTimeParseException) {
            throw UserException("Fecha incorrecta, usar $pattern", e)
        }
    }
}
```

```

class TimeCounterWindow : MainWindow<TimeCounter> {
    constructor(model: TimeCounter) : super(model)
    override fun createContents(mainPanel: Panel) {
        title = "Días"
        /*...*/
        Label(datePanel).setText("Ingresar fecha:")
        TextBox(datePanel)
            .bindValueToProperty<Any, ControlBuilder>("date")
            .setTransformer(DateTransformer())
        Label(datePanel).setText("Diferencia:")
        Label(datePanel)
            .bindValueToProperty<Any, ControlBuilder>("diffStr")
    }
}

```

Problemas de Transformers

- Es necesario verificar que el valor a transformar tengas las características deseadas
- El *Transformer* puede tener más responsabilidades de las necesarias

Podemos resolver los errores utilizando un objeto que filtre (*Filter*) la entrada, permitiendo ingresar exclusivamente los valores correctos (aunque no siempre lo resuelve del todo).

Filtros comunes:

- Sólo Números
- Sólo caracteres alfanuméricos
- Máximo X caracteres

```

class DateTextFilter : TextFilter {
    override fun accept(event: TextInputEvent): Boolean {
        val expected = listOf(
            "\\d", "\\d?", "/", "\\d", "\\d?", "/", "\\d{0,4}"
        )
        val regex = expected.reversed()
            .fold("") { accum, elem -> "($elem$accum)?" }
            .toRegex()
        return event.potentialTextResult.matches(regex)
    }
}

```

Filters ⇒ Uso directo

```

class TimeCounterWindow : MainWindow<TimeCounter> {
    constructor(model: TimeCounter) : super(model)
    override fun createContents(mainPanel: Panel) {
        title = "Días"
        /*...*/
        Label(datePanel).setText("Ingresar fecha:")
        TextBox(datePanel)
            .withFilter(DateTextFilter())
            .bindValueToProperty<Any, ControlBuilder>("date")
            .setTransformer(DateTransformer())
        /*...*/
    }
}

```

Filters ⇒ CustomBox

```
class DateBox(container: Panel) : TextBox(container) {
    override fun <M, C: ControlBuilder> bindValueToProperty(propertyName : String):
        Binding<M, Control, C> {
        val binding = super.bindValueToProperty<M, C>(propertyName)
        binding.setTransformer(DateTransformer())
        this.withFilter(DateTextFilter())
        return binding
    }
}

class TimeCounterWindow : MainWindow<TimeCounter> {
    override fun createContents(mainPanel: Panel) {
        title = "Días"
        /*...*/
        Label(datePanel).setText("Ingresar fecha:")
        DateBox(datePanel).bindValueToProperty<Any, ControlBuilder>("date")
        /*...*/
    }
}
```

Manejo de Colecciones

Existen varios componentes que permiten representar listados de datos:

- Selector (Dropdown)
- List
- Radio Selector
- Tables

Selectors ⇒ Adapters

Para mostrar un dato con formato complejo tenemos dos opciones:

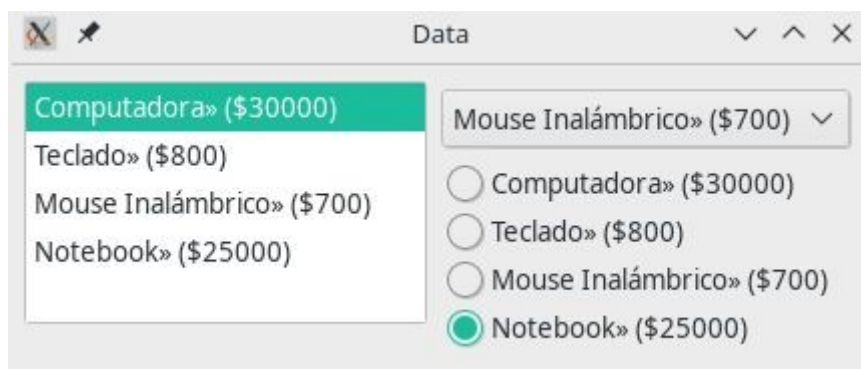
- Redefinir el `toString()` del modelo
- Setear un *Adapter*

Adapters ⇒ Selectors

```
class StoreWindow : MainWindow<Store> {  
    override fun createContents(mainPanel: Panel) {  
        mainPanel.setLayout(HorizontalLayout())  
        val prodList = List<Product>(mainPanel)  
        prodList.bindValueToProperty<...>("pc")  
        prodList.bindItemsToProperty("products")  
            .adaptWith(Product::class.java, "description")  
  
        val rightPanel = Panel(mainPanel)  
        val prodSelect = Selector<Product>(rightPanel)  
        prodSelect.bindValueToProperty<...>("mouse")  
        prodSelect.bindItemsToProperty("products")  
            .adaptWith(Product::class.java, "description")  
  
        val prodRadio = RadioSelector<Product>(rightPanel)  
        prodRadio.setWidth(200)  
        prodRadio.bindValueToProperty<...>("notebook")  
        prodRadio.bindItemsToProperty("products")  
            .adaptWith(Product::class.java, "description")  
    }  
}
```

```
@Observable class Store {  
    var products = listOf(  
        Product("Computadora", 30000),  
        Product("Teclado", 800),  
        Product("Mouse Inalámbrico", 700),  
        Product("Notebook", 25000)  
    )  
    var pc = products[0]  
    var mouse = products[2]  
    var notebook = products[3]  
}
```

```
@Observable class Product  
    (var name: String, var price: Int) {  
        fun description() = "$name» ($$price)"  
    }  
}
```



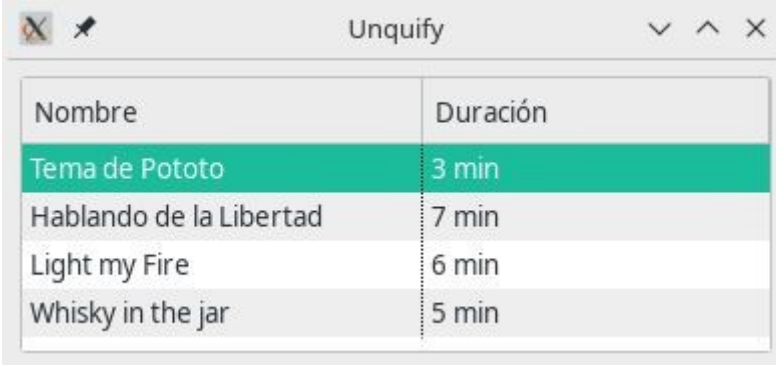
Tables

```
class PlaylistWindow : MainWindow<Playlist> {
    override fun createContents(mainPanel: Panel) {
        title = "Unquify"
        val songs = Table(mainPanel, Song::class.java)
        songs.setNumberVisibleRows(5)
        songs.bindItemsToProperty("songs")
        songs.bindValueToProperty<...>("selectedSong")
        Column<Song>(songs)
            .setTitle("Nombre")
            .setFixedSize(200)
            .bindContentsToProperty("name")

        Column<Song>(songs)
            .setTitle("Duración")
            .setFixedSize(150)
            .bindContentsToProperty("durationDescription")
    }
}
```

```
@Observable
class Playlist {
    val songs = listOf(
        Song("Tema de Pototo", 3),
        Song("Hablando de la Libertad", 7),
        Song("Light my Fire", 6),
        Song("Whisky in the jar", 5)
    )
    var selectedSong = songs[0]
}
```

```
@Observable
class Song(val name: String, val duration: Int) {
    fun durationDescription() = "$duration min"
}
```



The screenshot shows a window titled "Unquify" with a table containing four rows of song data. The first row, "Tema de Pototo", is highlighted in green. The table has two columns: "Nombre" and "Duración".

Nombre	Duración
Tema de Pototo	3 min
Hablando de la Libertad	7 min
Light my Fire	6 min
Whisky in the jar	5 min

//Ejemplo de búsqueda de nuestro TP:(Version sin adapter)

```
val productPanel = Panel(contentsPanel)
Label(productPanel)
    .setText("Administración De Productos")
    .setFontSize(25)
    .alignCenter()

val searchProductPanel = Panel(productPanel)
searchProductPanel.setLayout(HorizontalLayout())
    TextBox(searchProductPanel)
        .setWidth(200)
        .bindValueToProperty<Any, ControlBuilder>("productFilter")
        .setTransformer(NumericTransformer())

Button(searchProductPanel)
    .setCaption("Buscar")
    .onClick { modelObject.updateProductList() }
```

&& :: || :: **WEB** :: || :: &&

Protocolo HTTP (Hypertext Transfer Protocol)

- Protocolo de Capa de aplicación (nivel 7 del Modelo OSI) Comunicación para la transferencia de información en la WWW.
- Sin estado, es decir, no almacena información sobre conexiones anteriores.
- Orientado al esquema Petición-Respuesta entre un cliente y un servidor

Métodos de petición:

GET: Solo para recuperar información, parámetros deben ser enviados por URL.
POST: Envía datos para ser procesados, los datos deben ser enviados en el cuerpo (*body*) de la petición.
PUT: Sube, carga o realizar un “*upload*” de un recurso específico (file)
DELETE: Elimina un recurso específico

Códigos de Respuesta

200: OK (Petición correcta)
400: Bad Request (Petición incorrecta) el servidor no pudo interpretar la solicitud, por una sintaxis errónea.
404: Not Found (Recurso no encontrado)
500: internal Server Error (error no controlado) El servidor no puede controlar una excepción.

1xx	2xx	3xx	4xx	5xx
1xx: Mensaje informativo.	2xx Éxito	3xx: Redirección	4xx: Error del cliente	5xx: Error del servidor
	200: OK	300: Multiple Choice	<u>400</u> : Bad Request	500: Internal Server Error.
	201: Created	301: Moved Permanently	401: Unauthorized	501: Not Implemented
	202: Accepted	302: Found	403: Forbidden	502: Bad Gateway
	204: No Content	304: Not Modified ?	404: Not Found	503: Service Unavailable

API REST

Representational State Transfer

Cualquier interfaz entre sistemas que utilicen el protocolo HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como *XML* y *JSON*.

Stateless - Sin estado

- Utilizado en Aplicaciones WEB
- El protocolo *HTTP* es sin estado.
- El servidor no almacena peticiones anteriores
- Tras responder una petición, se cierra la conexión
- El servidor toma cualquier petición como una nueva.
- El servidor no retiene información de sesión
- No hay necesidad de liberar recursos tomados
- Información adicional del lado del cliente para determinar
- información de usuario o sesión (*Cookie* o *Token*).

Stateful - CON estado

- Utilizado en aplicaciones *Desktop* o protocolo *FTP*.
- El servidor mantiene información de sesión, con el objetivo de representar flujos de trabajos y estados.
- NO se persiste el estado, no se mantienen al reiniciar los servidores. Información volátil
- Diferentes peticiones pueden mantener datos compartidos. Se mantiene información del cliente a lo largo de varias invocaciones.
- Es necesario predecir la capacidad de *hardware*.
- Necesidad de liberar recursos ante el no uso prolongado.

CARACTERÍSTICAS	CON ESTADO	SIN ESTADO
Mantiene información de sesión	SI	NO
Diferentes peticiones pueden compartir información	SI	NO
Mantiene información volátil	NO	SI
Contexto y tamaño predecible	NO	SI
Necesidad de liberación de recursos (<i>garbage collector</i>)	SI	NO
Necesidad de reconstrucción ante caídas	SI	NO

API:

- Interfaz de Programación de Aplicaciones (Del inglés, *Application Programming Interface*)
- Conjunto de comandos, protocolos, funciones, objetos, etc...
- Provee estándares para facilitar la interacción con componentes
- Encapsula tareas complejas en otras más simples,

APIs Web:

- Implementan servicios
- Exponen recursos
- Permite interactuar con multiplicidad de tecnologías
- Establecen un protocolo de comunicación
- Agregan una capa de seguridad

REST:

- Transferencia de Estado Representacional
- **RE**presentational **S**tate **T**ransfer
- Estilo de arquitectura de software que provee estándares para la interacción entre sistemas web.
- Hace que la comunicación entre sistemas sea más simple

APIs RESTful:

- Aquellas aplicaciones que son compatibles con los principios **REST**:
- *Stateless*
- Arquitectura Cliente-Servidor
- Uso de Caché
- Interface Uniforme.

Formato de Intercambio

Se necesita un formato definido para intercambiar información. Los dos formatos más extendidos son: *JSON (JavaScript Object Notation Extensible Markup Language)* y *XML (Extensible Markup Language)*

La sintaxis de *JSON* deriva de la sintaxis de notación de objetos de *JavaScript*:

- Información como par "key": "value"
- Datos separados por coma (,)
- Las llaves ({}) contienen objetos
- Los corchetes ([]) contienen listas

Para cada *REQUEST*, en una *API REST* se define la estructura a la cual el cliente se debe ajustar para recuperar o modificar un recurso. En general consiste de:

- Verbo *HTTP*: define qué tipo de operación realizar
- Protocolo aceptado: *HTTP 1.1*, *HTTP 1.0*
- Media Data aceptada: *HTML*, *JSON*, *XML*
- Encabezado: (opcional) permite pasar información extra
- Ruta al recurso
- Cuerpo de mensaje (opcional) que contiene datos.

Por cada *REQUEST* que se recibe se debe retornar un *RESPONSE* con la información necesaria para describir lo que ocurrió:

- *HTTP Code* acorde a lo sucedido con la ejecución
- Protocolo de respuesta
- Media-data de la respuesta
- Cuerpo de mensaje (opcional) con la información requerida.

CRUD

El modelo debe poder crear, leer, actualizar y eliminar recursos (**Create, Read, Update, Delete**). A esto se le llama *CRUD*. Es la funcionalidad mínima que se espera de un modelo.

El paradigma *CRUD* es muy común en la construcción de aplicaciones web porque proporciona un modelo mental sobre los recursos, de manera que sean completos y utilizables.

Los CRUD se suelen armar respetando un estándar de URIs y métodos:

Crear	POST	/users
Leer (todos)	GET	/users
Leer (uno)	GET	/users/:id
Actualizar	PUT	/users/:id
Eliminar	DELETE	/users/:id

Muchas veces es necesario agregar información a la solicitud. Puede ser para filtrar una búsqueda o bien para que la respuesta incluya más o menos información.

Para estos casos se suelen utilizar parámetros de consulta (*query parameters*). Se escriben como un par **clave=valor** separados por **&**.

//EJEMPLO DE NUESTRO TP – API

```
/**/  
app.routes {  
  path("login") {  
    post(controller::login)  
  }  
  path("users") {  
    get(controller::getAllUsers)  
    path(":id") {  
      get(controller::findUser)  
      put(controller::updateUser)  
    }  
  }  
}
```

```

    delete(controller::deleteUser)
  }
  path("email") {
    path(":email") {
      get(controller::findUserByMail)
    }
  }
  path("register_form") {
    post(controller::addUserByForm)
  }
  path("register") {
    post(controller::addUser)
  }
}
/**/

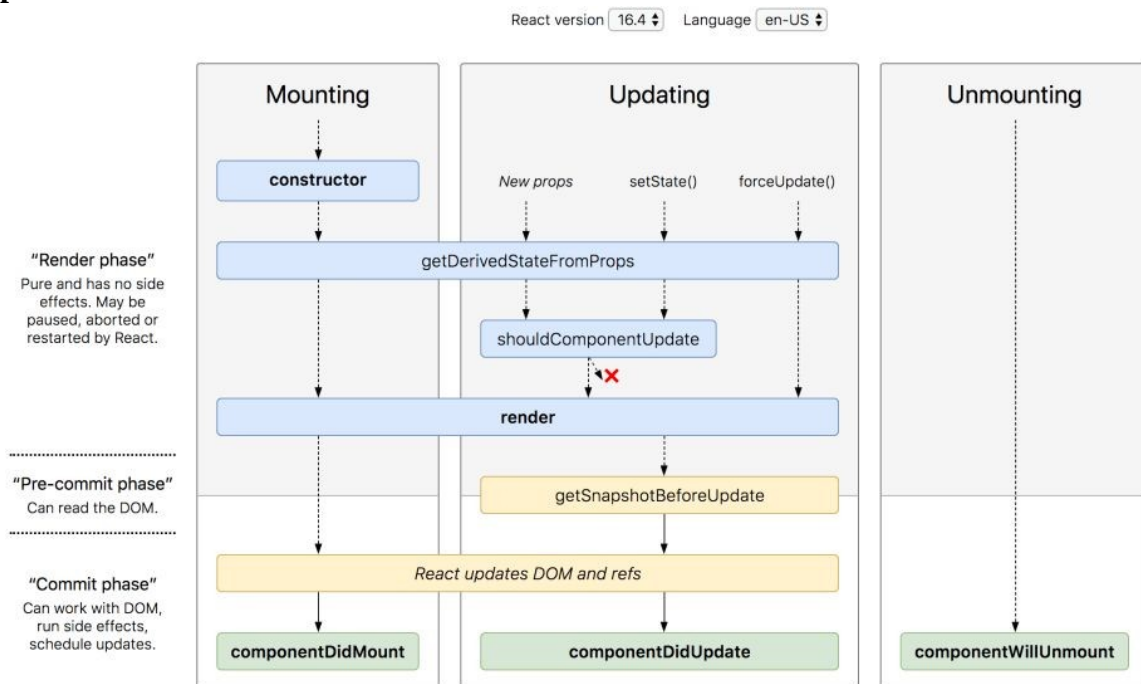
```

REACT

- Es una librería de *Javascript*.
- Se encuentra basado en componentes.
- Implementaron el “*Virtual DOM*”
- Defina la extensión *.JSX*
- Es una extensión en la que se puede escribir *HTML* junto al *javascript* sin tener que definirlo como un String.

LIFECYCLE

Operaciones con Colecciones



JAVASCRIPT:

== Débil
 === Estricta

Variables

- Mutables

```
var a = "Se está dejando de usar"
```

```
let a = "para usar let"
```

- Inmutables

```
const b = 1
```

Operación con colecciones:

```
const list = [1, 2, 3]
```

```
Filter list.filter( v => v > 2 ) // [3]
```

```
Map list.map(v => v * 2 ) // [2, 4, 6]
```

```
Every list.every(v => v > 2 ) // false
```

```
Find list.find(v => v > 3) // undefined
```

```
First, Last list[0] // 1
```

```
Sorted list.sort() // [1, 2, 3]
```

Funciones:

```
function sayHello(name) {  
  return `Hello ${name}`  
}
```

```
const sayHello = name => `Hello ${name}`
```

- Tiene default parameters

```
function sayHello(name = 'World') {  
  return `Hello ${name}`  
}
```

//EJEMPLO DE NUESTRO TP – REACT

//api.jsx – Pegandole a la API.

```
import axios from 'axios';
```

```
const server = 'http://localhost:7000';
```

```
const request = (type, path, body) => axios  
  .request({ url: `${server}${path}`, method: type, data: body })  
  .then(req => req.data);
```

```
export const signUp = body => request('post', '/users/register', body);  
export const signIn = body => request('post', '/login', body);  
export const getPendingOrdersFrom = body => request('get', '/orders_pending' + body);  
export const getHistoricOrdersFrom = body => request('get', '/order_historic/' + body);  
export const getMenu = body => request('get', '/restaurant/' + body);  
export const restaurants = body => request('get', '/restaurant/');  
export const menus = body => request('get', '/menus');  
export const products = body => request('get', '/products/');  
export const findRestaurant = body => request('get', '/findrestaurant/' + body);  
export const mySearch = body => request('get', '/search/' + body);
```

```

export default class App extends React.Component {
  render() {
    return (
      <BrowserRouter>
        <Switch>
          <Route exact path="/payorder" component={PayOrder} />
          <Route exact path="/sc" component={ShoppingCart} />
          <Route exact path="/restaurants" component={Restaurants} />
          <Route exact path="/orders" component={Orders} />
          <Route exact path="/users/register" component={SignUp} />
          <Route exact path="/signIn/:id" component={SignIn} />
          <Route exact path="/us" component={NavbarUs} />
          <Route exact path="/all_restaurants" component={NavbarRestaurants} />
          <Route exact path="/all_menus" component={NavbarMenues} />
          <Route exact path="/products" component={NavbarProducts} />
          <Route exact path="/contact" component={NavbarContact} />
          <Route exact path="/register" component={NavbarRegister} />
          <Route exact path="/home" component={Home} />
          <Route path="/" component={Home} />
        </Switch>
      </BrowserRouter>
    );
  }
}

```

//EJEMPLO DE NUESTRO TP – REACT

//app.jsx – Routeo

```

/**/
import NavbarRegister from './homeComponents/NavbarRegister';
import NavbarProducts from './homeComponents/NavbarProducts';
export default class App extends React.Component {
  render() {
    return (
      <BrowserRouter>
        <Switch>
          <Route exact path="/payorder" component={PayOrder} />
          <Route exact path="/sc" component={ShoppingCart} />
          <Route exact path="/restaurants" component={Restaurants} />
          <Route exact path="/orders" component={Orders} />
          <Route exact path="/users/register" component={SignUp} />
          <Route exact path="/signIn/:id" component={SignIn} />
          <Route exact path="/us" component={NavbarUs} />
          <Route exact path="/all_restaurants" component={NavbarRestaurants} />
          <Route exact path="/all_menus" component={NavbarMenues} />
          <Route exact path="/products" component={NavbarProducts} />
          <Route exact path="/contact" component={NavbarContact} />
          <Route exact path="/register" component={NavbarRegister} />
          <Route exact path="/home" component={Home} />
          <Route path="/" component={Home} />
        </Switch>
      </BrowserRouter>
    );
  }
}

```

```

        </Switch>
      </BrowserRouter>
    );
  }
}

```

//EJEMPLO DE NUESTRO TP – REACT

//home.jsx – Render

```

render() {
  return (
    <div>
      <Navbar />
      <Body />
      <Footer />
      {this.state.error && <div>{this.state.error}</div>}
    </div>
  );
}

```

//OrderForm.jsx

```

class OrderForm extends React.Component{
  constructor(){
    super();
    this.state={
      code: 0,
      username: "",
      state: 'Pending',
      restaurant:"",
      value:"",
      menu:""
    }
    this.handleInput = this.handleInput.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

```

```

  handleSubmit(e) {
    e.preventDefault();
    this.props.onAddTodo(this.state);
    this.setState({
      code: "",
      username: "",
      restaurant: "",
      state: 'Pending',
      value: "",
      menu:""
    });
  }

```

```

  handleInput(e){
    const { value, name } = e.target;
    this.setState({

```

```

    [name]: value
  })
}

```

```

render(){
  return (
    <div className="card mr-4 md-4 mt-4">
      <form onSubmit={this.handleSubmit} className="card-body">
        <div className="form-group">
          <input
            type="text"
            name="id"
            onChange={this.handleInput}
            className="form-control"
            placeholder="Username"
          />
        </div>
        <div className="form-group">
          <input
            type="text"
            name="value"
            className="form-control"
            placeholder="Value"
            onChange={this.handleInput}
          />
        </div>
        <div className="form-group">
          <select
            name = "restaurant"
            className="form-control"
            onChange={this.handleInput}>
            <option>"La Conga"</option>
            <option>"Los Maizales"</option>
            <option>"Güerrin"</option>
            <option>"El Tano"</option>
          </select>
        </div>
        <div className="form-group">
          <select
            name = "menu"
            className="form-control"
            onChange={this.handleInput}>
            <option>"Menu0"</option>
            <option>"Menu1"</option>
            <option>"Menu2"</option>
            <option>"Menu3"</option>
          </select>
        </div>
        <button type="submit" className="btn btn-primary">
          Enviar!
        </button>
      </form>
    </div>
  )
}

```

```

        </form>
      </div>
    )
  }
};
export default OrderForm;

```

//Body.jsx

```

import React from 'react';

import './css/Body.css';
import SignUp from './SignUp';
import SignIn from './SignIn';
import imag1 from './images/img1.jpg';

export default class Body extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      renderReg: false,
      renderLog: false
    };
    this.handlerReg = this.handlerReg.bind(this);
    this.handlerLog = this.handlerLog.bind(this);
  }

  register = () => {
    this.setState({renderReg : !this.state.renderReg})
    if(this.state.renderLog){
      this.setState({renderLog : false})
    }
  }

  login = () => {
    this.setState({renderLog : !this.state.renderLog})
    if(this.state.renderReg){
      this.setState({renderReg : false})
    }
  }

  handlerReg() {
    this.setState({ renderReg: false })
  }
  handlerLog() {
    this.setState({ renderLog: false })
  }
}

```

//Body.jsx - Render

```
render() {
  return (
    <div>
      /**/
      <a href="#3" className="btn btn-1 btn-1b" onClick={e => this.register()}
    >Registrarte</a>
      <a href="#2" className="btn btn-2 btn-2b" onClick={() => this.login()}
    >Loguearte</a>
      { this.state.renderReg &&
        <SignUp id = "3" handlerReg = {this.handlerReg} /> }
      { this.state.renderLog &&
        <SignIn id="2" handlerLog = {this.handlerLog} /> }
    </div>
    { /*welcome*/ }
  /**/
  )
}
```

//NAVBAR genera NAVBAR ITEMS

```
<div className="collapse navbar-collapse" id="navbarTogglerDemo01">
  <a className="navbar-brand" href="/">Home</a>
  <ul className="navbar-nav mr-auto mt-2 mt-lg-0">
    {items.map(function(currentValue, index, array){
      return <NavBarItem key={currentValue.code}
        name={currentValue.name}
        id={currentValue.id}
        isFirstOne={index==0? true : false}/>;
    })}
  </ul>
</div>
```

//Page.jsx – Render

```
render() {
  return (
    <div>
      <Navbar />
      <Container content={this.props.child} id={this.props.id}/>
      <Footer />
    </div>
  );
}
```

//Containner.jsx – Dentro del render

```
/**/
{this.props.id === "0" && <ContentUs />}
```



```

    {this.props.id === "1" && <ContentRestaurant k={this.props.content}/>}
    {this.props.id === "2" && <ContentMenu k={this.props.content}/>}
    {this.props.id === "3" && <ContentProduct k={this.props.content}/>}
    {this.props.id === "4" && <ContentContact k={this.props.content}/>}
    {this.props.id === "5" && <ContentRegister/>}

```

//Restaurants.jsx – Render

```

render() {
  const myRestos = this.props.k.map((resto, i) => {
    return(
      <div className="card mt-4 col-md-4" key={i}>
        <div className="card-headercard-title text-center">
          <h4>Restaurant: {resto.name}</h4>
          <span className="badge-pill badge-danger ml-2">
            {"open"}
          </span>
        <div className="card-body">
          {"Descripción: " + resto.description}
          <mark>{"Dirección: " + resto.address}</mark>
          <mark>{resto.code}</mark>
        </div>
      </div>
    )
  })
}

```