

## Arquitectura MVC con TypeScript

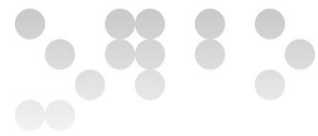
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Todo App</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <div id="root"></div>
    <script src="dist/bundle.js"></script>
  </body>
</html>
```

### **Modelo (todo.model.ts)**

La primera clase que se construye en este ejemplo es el modelo de la aplicación, `todo.model.ts`, la cual consiste en los atributos de la clase, y un método privado que está generando ID aleatorias (estas id's podrían venir del servidor desde una base de datos en un punto más avanzado de la aplicación). Los atributos son del siguiente tipo:

```
- id: number;
- text: string;
- complete: boolean;
```

Observa que tienes que definir también el valor por defecto de los atributos en el constructor. A continuación, se muestra la clase en `.js` previa a transformarla a TypeScript



```
/**
 * @class Model
 *
 * Manages the data of the application.
 */
class Todo {
  constructor({ text, complete } = { complete: false }) {
    this.id = this.uuidv4();
    this.text = text;
    this.complete = complete;
  }

  uuidv4() {
    return ([1e7] + -1e3 + -4e3 + -8e3 + -1e11).replace(/[018]/g, c =>
      (
        c ^
        (crypto.getRandomValues(new Uint8Array(1))[0] & (15 >> (c / 4)))
      ).toString(16)
    );
  }
}
```

### Servicio (todo.service.ts)

Las operaciones que se realizan sobre los TODOs son llevadas a cabo en el servicio. En este caso concreto utilizaremos un *array* para almacenar todos los TODOs (es decir, el *array* `todos` deberá contener objetos de la clase `Todo`) y construiremos los 4 métodos asociados a leer, modificar, crear y eliminar TODOs.

A continuación se muestra la clase `TodoService` que deberás transformar a TypeScript (Ten en cuenta que los *callback* son parámetros del tipo **Function**).

```
class TodoService {
  constructor() {
    this.todos = (JSON.parse(localStorage.getItem("todos")) || []).map(
      todo => new Todo(todo)
    );
  }
}
```



```
bindTodoListChanged(callback) {
  this.onTodoListChanged = callback;
}

_commit(todos) {
  this.onTodoListChanged(todos);
  localStorage.setItem("todos", JSON.stringify(todos));
}

addTodo(text) {
  this.todos.push(new Todo({ text }));

  this._commit(this.todos);
}

editTodo(id, updatedText) {
  this.todos = this.todos.map(todo =>
    todo.id === id
      ? new Todo({
        ...todo,
        text: updatedText
      })
      : todo
  );
  this._commit(this.todos);
}

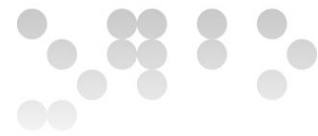
deleteTodo(_id) {
  this.todos = this.todos.filter(({ id }) => id !== _id);

  this._commit(this.todos);
}

toggleTodo(_id) {
  this.todos = this.todos.map(todo =>
    todo.id === _id ? new Todo({ ...todo, complete: !todo.complete }) : todo
  );

  this._commit(this.todos);
}
}
```

Vista (todo.views.ts)



La clase correspondiente a la vista es la encargada de gestionar el DOM de la aplicación.

Si observas el fichero relativo a la vista, `todo.views.js`, se han seleccionado elementos del árbol del DOM (página web) para asignarle eventos, las acciones de *click* en el botón borrar, sobre el texto, o el checkbox. Si observas el código (p.ej. línea 102), podrás ver que las acciones o *handles* son funciones pasadas como parámetros a la vista. De este modo, se ha conseguido abstraer completamente la vista de las acciones que se deben realizar. Las funciones de la vista son llamadas desde el controlador.

Para convertir este código en TypeScript debes tener en cuenta que los elementos html tienen un interface específico `HTMLElement` (<http://definitelytyped.org/docs/flipsnap—flipsnap/interfaces/htmllement.html>).

El fichero `.js` que debes ser transformado a TypeScript es el siguiente.

```
/**
 * @class View
 *
 * Visual representation of the model.
 */
class TodoView {
  constructor() {
    this.app = this.getElement("#root");
    this.form = this.createElement("form");
    this.input = this.createElement("input");
    this.input.type = "text";
    this.input.placeholder = "Add todo";
    this.input.name = "todo";
    this.submitButton = this.createElement("button");
    this.submitButton.textContent = "Submit";
    this.form.append(this.input, this.submitButton);
    this.title = this.createElement("h1");
    this.title.textContent = "Todos";
    this.todoList = this.createElement("ul", "todo-list");
    this.app.append(this.title, this.form, this.todoList);

    this._temporaryTodoText = "";
    this._initLocalListeners();
  }

  get _todoText() {
    return this.input.value;
  }
}
```



```
}

_resetInput() {
  this.input.value = "";
}

createElement(tag, className) { ... }

getElement(selector) { ... }

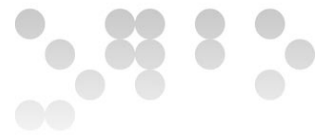
displayTodos(todos) { ... }
_initLocalListeners() { ... }

bindAddTodo(handler) { ... }

bindDeleteTodo(handler) { ... }

bindEditTodo(handler) { ... }

bindToggleTodo(handler) { ... }
}
```



### Controlador (`todo.controller.ts`)

Finalmente, la clase que une todas las partes es el controlador, donde se reciben las instancias de servicio y vista para crear los manejadores (es decir, las funciones encargadas de realizar una acción cuando se pulsa un botón o realizar una acción concreta en la vista).

En el controlador se han unido (*binding*) las acciones de la vista con las del servicio. El fichero `controller.js` que debe ser transformado a TypeScript es el siguiente:

```
/**
 * @class Controller
 *
 * Links the user input and the view output.
 *
 * @param model
 * @param view
 */
class TodoController {
  constructor(service, view) {
    this.service = service;
    this.view = view;

    // Explicit this binding
    this.service.bindTodoListChanged(this.onTodoListChanged);
    this.view.bindAddTodo(this.handleAddTodo);
    this.view.bindEditTodo(this.handleEditTodo);
    this.view.bindDeleteTodo(this.handleDeleteTodo);
    this.view.bindToggleTodo(this.handleToggleTodo);

    // Display initial todos
    this.onTodoListChanged(this.service.todos);
  }

  onTodoListChanged = todos => {
    this.view.displayTodos(todos);
  };

  handleAddTodo = todoText => {
    this.service.addTodo(todoText);
  };

  handleEditTodo = (id, todoText) => {
```



```
this.service.editTodo(id, todoText);  
};  
  
handleDeleteTodo = id => {  
  this.service.deleteTodo(id);  
};  
  
handleToggleTodo = id => {  
  this.service.toggleTodo(id);  
};  
}
```

### Lanzadora (app.ts)

El último paso que queda para hacer funcionar la aplicación es “lanzarla”. Para eso tenemos el fichero `app.ts` que actúa de lanzador de toda la lógica a través de la instanciación de `Controller`.

```
const app = new TodoController(new TodoService(), new TodoView());
```