Manual Técnico – Sistema Tenis Master

Versión: 1.0

Fecha: Noviembre 2025

Equipo de desarrollo: RedSoft

1. Introducción general

Tenis Master es una aplicación web desarrollada en Node.js y Express para la gestión de reservas de canchas de tenis, administración de personas usuarias (clientes/jugadores, empleados y administradores), registro de pagos y búsqueda de jugadores compatibles ("macheo").

El sistema está orientado a clubes o complejos deportivos que necesitan digitalizar el proceso de reserva de turnos, evitar solapamientos de horarios, llevar un registro de los pagos y disponer de un historial de reservas que sirva como referencia administrativa.

Este manual técnico está dirigido a desarrolladores y personas con conocimientos en programación web que necesiten instalar, mantener o extender la aplicación. Se describe la arquitectura, los componentes principales, la base de datos, la configuración del entorno, el esquema de sesiones con connect-mongo y el proceso de despliegue en Render.

2. Arquitectura del sistema

2.1 Estilo arquitectónico

El sistema adopta una arquitectura MVC ligera (Modelo–Vista–Controlador) sobre Express:
• Modelos (M) definidos con Mongoose en la carpeta models/.
• Controladores (C) en la carpeta controllers/ encapsulan la lógica de negocio.
• Vistas (V) implementadas con el motor de plantillas Pug en la carpeta views/.
• Las rutas Express en la carpeta routes/ actúan como capa de enrutamiento entre HTTP y los controladores.

La aplicación es monolítica: un único servidor Node.js gestiona tanto el backend como el renderizado de vistas. Los archivos estáticos (imágenes, estilos) se sirven desde la carpeta public/.

2.2 Tecnologías principales

Backend:
• Node.js 18+.
• Express 4.x como framework HTTP.

• express-session para manejo de sesiones.
• connect-mongo como store de sesiones persistente en MongoDB.
• Mongoose 8.x como ODM para MongoDB.
• bcrypt para hash de contraseñas.
• dotenv para gestión de variables de entorno.

Frontend:
• Pug como motor de plantillas.
• TailwindCSS (vía CDN) para estilos utilitarios.
• Imágenes y recursos estáticos en public/.

Infraestructura y base de datos:
• MongoDB Atlas (cluster en la nube).
• Despliegue de la aplicación en Render como Web Service.

3. Estructura del proyecto

La estructura de carpetas contenida en Tenis_master.zip es la siguiente (resumen):

• index.js → Punto de entrada de la aplicación.
• package.json → Dependencias y scripts.
• .env → Variables de entorno (no se versiona).

• controllers/ → Controladores de negocio.
  – panelController.js → Lógica del panel principal según rol.
  – reservaController.js → Lógica de reservas, pagos e historial.
  – usuarioController.js → Autenticación y gestión de usuarios.

• middlewares/
  – autorizar.js → Middleware de login y autorización por rol.

• models/
  – Usuario.js → Esquema de usuarios/clientes.
  – Reserva.js → Esquema de reservas de canchas.
  – Pago.js → Esquema de pagos asociados a reservas.

• routes/
  – appRoutes.js → Rutas de login, logout y panel.
  – reservaRoutes.js → Rutas de reservas, pagos e historial.
  – usuarioRoutes.js → Rutas de CRUD de usuarios y registro.
  – macheoRoutes.js → Rutas de macheo de jugadores.

• views/ → Plantillas Pug.

– login.pug, registro.pug, panel.pug.
– panel-reservas-admin.pug, panel-reservas-cliente.pug.
– historial-reservas-admin.pug.
– reservar.pug, pago.pug, macheo.pug, usuarios.pug.
– error-autorizacion.pug, error-login.pug.

• public/
– images/ → Imágenes de fondo y logotipos.

4. Base de datos y modelos de dominio

4.1 Modelo Usuario

El modelo Usuario (models/Usuario.js) representa tanto a administradores y empleados como a clientes/jugadores. Se almacena en la colección usuarios y posee, entre otros, los siguientes campos:

• username (String, requerido, único): correo o identificador de login.
• password (String, requerido): contraseña hasheada con bcrypt.
• rol (String): 'administrador', 'empleado' o 'cliente'.
• nombre, apellido (String): datos personales básicos.
• celular (String, opcional).
• nivel (String): nivel de juego del jugador (para macheo).
• disponibilidad (Array embebido): lista de objetos con dia, desde, hasta, utilizados para filtrar jugadores compatibles.

El esquema utiliza timestamps para registrar createdAt y updatedAt automáticamente.

4.2 Modelo Reserva

El modelo Reserva (models/Reserva.js) modela una reserva de cancha:

• jugadorId (ObjectId, ref Usuario, requerido): persona jugadora a la que pertenece la reserva.
• jugadorEmail (String): redundancia del correo para facilitar reportes.
• fecha (Date, requerido): día de la reserva normalizado a las 00:00h.
• horaInicio (String, requerido, formato HH:MM).
• horaFin (String, requerido, formato HH:MM).
• cancha (String, requerido): identificador lógico de la cancha.
• estado (String, enum ['activa', 'pagado', 'cancelada']): estado actual de la reserva.
• createdBy (ObjectId, ref Usuario): usuario que creó la reserva.

Este modelo es la base para el panel de reservas, el cálculo de importe y el historial.

4.3 Modelo Pago

El modelo Pago (models/Pago.js) almacena la información de los pagos realizados:

- reservaId (ObjectId, ref Reserva, requerido): referencia a la reserva pagada.
- clienteId (ObjectId, ref Usuario): jugador que realiza el pago.
- metodo (String): 'tarjeta', 'efectivo', etc.
- importe (Number): monto cobrado.
- numeroTarjetaEnmascarado (String): últimos 4 dígitos, cuando aplica.
- fechaPago (Date, default ahora).
- registradoPor (ObjectId, ref Usuario): empleado o admin que registró el pago.

El almacenamiento de datos sensibles se minimiza: no se guarda el número completo de tarjeta ni el CVV.

5. Configuración e instalación

5.1 Dependencias y scripts

El archivo package.json define las dependencias del proyecto:

- express, mongoose, pug.
- express-session, connect-mongo.
- bcrypt, dotenv.
- nodemon (como dependencia de desarrollo).

Scripts relevantes:
- npm start → ejecuta node index.js.
- npm run dev → ejecuta nodemon index.js (recarga automática en desarrollo).

5.2 Variables de entorno (.env)

Las siguientes variables deben declararse en el archivo .env (no se versiona):

- MONGO_URI → cadena de conexión a MongoDB Atlas.
- PORT → puerto HTTP (ej. 3000).
- CANCHA_PRECIO_POR_HORA → precio de alquiler por hora (Number).
- SESSION_SECRET → secreto utilizado por express-session.

Ejemplo:
MONGO_URI=mongodb+srv://usuario:password@cluster/.../tenis_master
PORT=3000
CANCHA_PRECIO_POR_HORA=24000
SESSION_SECRET=un-secreto-seguro

5.3 Pasos de instalación en entorno local

1. Clonar el repositorio con el código fuente.
2. Crear el archivo .env con las variables necesarias.
3. Ejecutar npm install para instalar dependencias.
4. Iniciar el servidor con npm start.
5. Abrir el navegador en http://localhost:3000/.

6. Componentes principales del backend

6.1 index.js

El archivo index.js es el punto de entrada de la aplicación. Sus responsabilidades principales son:

• Importar y configurar Express.
• Cargar dotenv para leer las variables de entorno.
• Conectar a MongoDB mediante mongoose.connect(MONGO_URI).
• Configurar las sesiones con express-session y connect-mongo.
• Registrar los middlewares globales (urlencoded, json, static, res.locals.usuario).
• Configurar el motor de vistas Pug.
• Montar las rutas definidas en la carpeta routes/.
• Ejecutar al inicio la función cancelarReservasVencidas(), que marca como canceladas las reservas activas con fecha anterior a hoy.

La configuración de sesiones se realiza del siguiente modo (resumen conceptual):

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  store: MongoStore.create({
    mongoUrl: process.env.MONGO_URI,
    collectionName: 'sessions'
  }),
  cookie: { maxAge: 1000 * 60 * 60 * 2 }
}));
```

De este modo las sesiones quedan persistidas en la base de datos MongoDB en la colección sessions, evitando el uso de MemoryStore.

6.2 Middlewares de autenticación y autorización

El middleware definido en middlewares/autorizar.js provee dos funciones clave:

• requerirLogin(req, res, next): verifica que exista req.session.usuario. Si no hay sesión, redirige a la página de login.
• autorizar(rolesPermitidos): devuelve un middleware que comprueba que el rol de la persona usuaria se encuentre dentro de la lista rolesPermitidos. En caso contrario, responde con un 403 y renderiza error-autorizacion.pug.

Estos middlewares se usan en las rutas para restringir el acceso según rol (administrador, empleado, cliente).

6.3 Controladores de usuario

El controlador usuarioController.js implementa:

• mostrarLogin(): renderiza el formulario de login, o redirige al panel si ya existe sesión.
• login(): valida las credenciales contra la base de datos utilizando bcrypt.compare y, en caso exitoso, almacena un objeto usuario simplificado en req.session.usuario.
• logout(): destruye la sesión y redirige al login.
• obtenerUsuarios(), crearUsuario(), actualizarUsuario(), eliminarUsuario(): operaciones de mantenimiento sobre clientes cuando el rol es administrador o empleado.
• registrarCliente(): alta pública de usuarios con rol=cliente, utilizada por la vista de registro.

6.4 Controlador de reservas y pagos

El controlador reservaController.js concentra la lógica de negocio relacionada con reservas y pagos. Algunas funciones clave son:

• mostrarFormularioReserva(): carga el formulario de reserva, prellenando información cuando viene desde macheo.
• crearReserva(): valida datos obligatorios, impide reservas en fechas/horas pasadas, normaliza la fecha y comprueba solapamientos por cancha antes de crear una nueva reserva.
• listarReservas(): listado general (para administración) usando filtros opcionales.
• actualizarReserva() y eliminarReserva(): permiten modificar o eliminar reservas según reglas del negocio.
• panelReservas(): devuelve una vista distinta según el rol:
  – Cliente: panel-reservas-cliente.pug con sus propias reservas.
  – Admin/Empleado: panel-reservas-admin.pug mostrando solo reservas actuales y futuras.
• mostrarPago() y pagarReserva(): flujo de pago donde se calcula el importe según la duración (en bloques de 30 minutos) y se actualiza el estado de la reserva a pagado.

• historialReservas(): obtiene todas las reservas con fecha anterior a hoy y las renderiza en historial-reservas-admin.pug para administradores y empleados.

6.5 Macheo de jugadores

La ruta /macheo y el archivo macheoRoutes.js permiten buscar jugadores compatibles. Se filtra por:

• rol='cliente'.
• disponibilidad no vacía.
• nivel (opcional) y texto libre (q) sobre nombre, apellido o username.

Los resultados se muestran en la vista macheo.pug, y desde allí es posible iniciar una reserva preseleccionando al jugador.

7. Gestión de errores y validaciones

El sistema implementa varias capas de validación y manejo de errores:

• Validación de campos obligatorios (fecha, horaInicio, horaFin, cancha, jugador) antes de crear reservas.
• Validación temporal: no se permiten reservas con fecha/hora de inicio en el pasado.
• Validación de solapamientos: para una misma cancha y día, se consulta si existe alguna reserva activa cuyo rango horario se solape con el solicitado.
• Validaciones en el flujo de pago: se comprueba que la reserva exista, que esté activa y que se reciba un método de pago válido.
• Manejo de errores de autenticación y autorización: redirecciones al login o vistas de error-autorizacion.
• Manejo de errores generales con bloques try/catch en los controladores; en caso de excepción se registra en consola y se muestra un mensaje genérico a la persona usuaria.

8. Pruebas y calidad

El proyecto no incorpora por el momento pruebas automáticas (unitarias o de integración). Las verificaciones se realizan de manera manual mediante la interfaz web:

• Alta, login y logout de usuarios de diferentes roles.
• Creación de reservas con distintos escenarios (horarios válidos, horarios pasados, solapamientos, distintas canchas).
• Registro de pagos y verificación del cambio de estado de la reserva a pagado.
• Verificación de la lógica de limpieza de reservas vencidas al iniciar el servidor.
• Validación del listado de historial de reservas y panel de reservas futuras.
• Pruebas del módulo de macheo de jugadores.

Como mejora futura se recomienda incorporar Jest o Mocha para pruebas unitarias, Supertest para pruebas de rutas HTTP y Cypress o Playwright para pruebas end-to-end.

9. Despliegue en Render

El despliegue del sistema se realiza en la plataforma Render como un Web Service Node.js. El flujo recomendado es el siguiente:

1. Subir el repositorio de Tenis Master a GitHub.
2. Crear un nuevo Web Service en Render seleccionando el repositorio.
3. Configurar el runtime como Node y el comando de inicio npm start.
4. Definir las variables de entorno (ENV VARS) en la interfaz de Render:
   – MONGO_URI
   – PORT
   – CANCHA_PRECIO_POR_HORA
   – SESSION_SECRET
5. En MongoDB Atlas, permitir el acceso desde la IP pública de Render o, para entornos de prueba, usar 0.0.0.0/0 en la IP Access List.
6. Render descargará el código, instalará las dependencias y ejecutará npm start, exponiendo una URL pública como https://master-tenis.onrender.com/.

10. Mantenimiento y extensiones futuras

Para extender el sistema Tenis Master se recomienda:

• Mantener la organización MVC actual al agregar nuevas funcionalidades.
• Crear nuevos modelos en models/, sus controladores en controllers/ y rutas en routes/.
• Reutilizar los middlewares de requerirLogin y autorizar para proteger las nuevas rutas.
• Actualizar las vistas Pug respetando la estética existente basada en Tailwind.
• Documentar cualquier cambio de esquema en los modelos de Mongoose y, si es necesario, preparar scripts de migración.
• Evaluar la incorporación de paginación en el historial de reservas para mejorar el rendimiento con grandes volúmenes de datos.

Este manual proporciona una visión técnica detallada de la implementación actual, sirviendo como base para tareas de mantenimiento, resolución de incidencias y ampliación funcional del sistema.