UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 14

DEVELOPING A WEB APP WITH EXPRESS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://docenti.unina.it/luigiliberolucio.starace

PREVIOUSLY, ON WEB TECHNOLOGIES

We learned a good deal about **Web Frameworks** and **Express**

- Routers, Middleware, Templating…

Let’s put our knowledge to good use and build a proper web app!

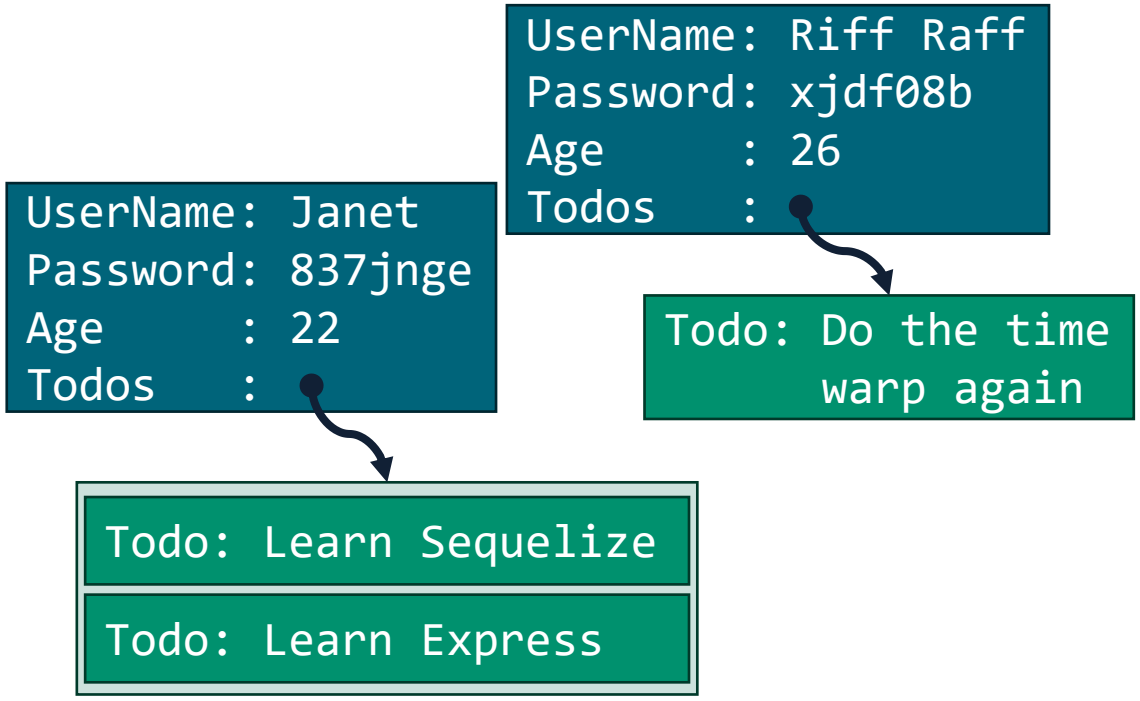
- We’ll re-implement our To-do List web app using Express (adding more features along the way)
- We’ll also manage persistency in a more advanced way, using an **Object-Relational Mapping** library

OBJECT–RELATIONAL MAPPING

OBJECT–RELATIONAL MAPPING (ORM)

When developing Object-Oriented software that persists data in a RDBMS, we deal with **two different representations** of our data

- One is objects living in the **heap**, the other data in **RDBMS tables**



Users table

Id	UserName	Password	Age
1	Janet	837jnge	22
2	Riff Raff	xjdf08b	26
4	Frank	K99fid8f	56

Todos table

Id	UserId	Todo
1	1	Learn Sequelize
2	1	Learn Express
4	2	Do the time warp again

OBJECT–RELATIONAL MAPPING (ORM)

- These two representations are quite different
- We need to keep both representations **aligned**
  - Changes to the domain objects need to be mapped to the RDBMS schema
  - Changes to the RDBMS schema might impact existing code
- Good amount of work and (repetitive) boilerplate code needed

ORM LIBRARIES

ORM libraries are ready-made solutions to support developers in keeping object-based and relational representations aligned

- Allow developers to work with high-level **abstractions** instead of SQL queries and statements
- Increase **productivity** by reducing the need for boilerplate code
- Make switching to different RDBMS even easier

ORM LIBRARIES: FEATURES

ORM libraries typically allow developers to:

- **Declaratively** define how database tables are mapped to objects
- **Automatically create** and **update** the **RDBMS schema** as needed
- Easily perform Create, Read, Update, Delete (**CRUD**) operations
- Compose and execute complex queries in a declarative way
- Manage transactions
- Support caching strategies

ORM LIBRARIES: EXAMPLES

Well-known ORM libraries include:

- Hibernate (Java)
- Entity Framework (C# - .NET)
- SQLAlchemy (Python)
- Propel (PHP)
- Active Record (Ruby on Rails)
- Sequelize, Prisma, TypeORM (Node.js)

Opinionated frameworks sometimes also include an ORM

SEQUELIZE

- Sequelize is an ORM for Node.js
- Supports Oracle, Postgres, MySQL, MariaDB, SQLite, SQL Server and more!
- Elegant Promise-based API
- The following slides are basedon Sequelize v. 6

INSTALLING SEQUELIZE

- Sequelize can be installed via **npm** as usual

```
@luigi → express-hello-world $ npm install sequelize
```

- You also need to install the drivers for the RDBMSs you plan to use:

```
@luigi → express-hello-world $ npm install pg pg-hstore # Postgres
@luigi → express-hello-world $ npm install mysql2
@luigi → express-hello-world $ npm install mariadb
@luigi → express-hello-world $ npm install sqlite3
@luigi → express-hello-world $ npm install tedious      # MS SQL Server
@luigi → express-hello-world $ npm install oracledb
```

- We'll use SQLite for the sake of simplicity

CONNECTING TO A DATABASE

```
import { Sequelize } from "sequelize";

//create connection
const database = new Sequelize("sqlite:mydb.sqlite");
//const database = new Sequelize('postgres://user:pass@example.com:5432/dbname')

try {
  await database.authenticate();
  console.log('Connection has been established successfully.');
} catch (error) {
  console.error('Unable to connect to the database:', error);
}
```

- Above code automatically creates a SQLite DB in the **mydb.sqlite** file
- Many different ways to connect to a database: docs

DEFINING MODELS

- Models are the essence of any ORM
- In Sequelize, Models are classes that extend Model
- Two equivalent ways to define models:
  - Calling the **define** method on a database connection
  - Creating a class that extends Model, and then calling its **init** method

DEFINING MODELS

```
import { DataTypes, Sequelize } from "sequelize";

//connection to database omitted

const User = database.define('User', { //Model attributes are defined here
  firstName: {
    type: DataTypes.STRING,
    allowNull: false
  },
  age: {
    type: DataTypes.INTEGER //allowNull defaults to true
  },
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  }
});
```

MODEL SYNCHRONIZATION

- When defining a Model, we're telling Sequelize about our data
- However, we start with a completely empty database
- We need to tell Sequelize to align the database schema with the models we defined
  - What if there exists no table to persist our Models?
  - What if some tables exist, but have different columns, less columns, or some other difference?
- Alignment is performed by Sequelize via **model synchronization**

MODEL SYNCHRONIZATION

- To syncronize a model with the database, we can call **model.sync(…)**
- It’s an async function, returning a Promise

```
const User = database.define('User', {/* Model specification omitted */});

User.sync().then( () => {
  console.log("Users table synchronized.")
}).catch( err => {
  console.err("Synchronization error:", err.message)
});
```

```
Executing: SELECT name FROM sqlite_master WHERE type='table' AND name='Users';
Executing: CREATE TABLE IF NOT EXISTS `Users` (`firstName` VARCHAR(255) NOT NULL,
`age` INTEGER, `id` INTEGER PRIMARY KEY AUTOINCREMENT, `createdAt` DATETIME NOT
NULL, `updatedAt` DATETIME NOT NULL);
Users table synchronized.
```

MODEL SYNCHRONIZATION

A few notes:

- Sequelize **inferred** a table name for our model (***Users***). By default, Sequelize uses the **pluralized** model name to name tables in the RDBMS. We can override this behaviour and specify a custom table name (see docs).
- From the logs, we can see that Sequelize checked whether the *Users* table existed (`SELECT name FROM sqlite_master WHERE type='table' AND name='Users';`), and then created the table.
- If we run the script again, it won't create the table since it already exists

MODEL SYNCHRONIZATION

- Sequelize supports different synchronization modes

```
User.sync();
//creates the table if not exists (does nothing if exists)

User.sync({force: true});
//re-creates the table, drops it if exists (destructive!)

User.sync({alter: true});
//checks the state of the table (columns, types, etc..) and alters it to match
the model (if needed)
```

- It is possible to synchronize multiple models at once by calling **sync()** on the database connection

CREATING ENTITIES

```
let janet = new User({ //We can first create an instance of a Model...
  name: "Janet",
  age: 22
});

await janet.save();  //...and then save it to the database
console.log("Janet saved to database.");

let riff = await User.create({  //Or we can use the static method Model.create()
  name: "Riff Raff",
  age: 26
});
console.log(`Riff Raff saved to database with id ${riff.id}.`);
```

DEFINING MODELS (USING CLASSES)

```
const User = database.define('User', {/* models specs */});

class Todo extends Model {}

Todo.init({
  todo: { type: DataTypes.TEXT, allowNull: false },
  done: { type: DataTypes.BOOLEAN, defaultValue: false },
  id:   { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true }
},{
  sequelize: database, modelName: "Todo"
});

database.sync().then( () => {console.log("Database synchronized.")});
```

DEFINING ASSOCIATIONS

- Sequelize supports one-to-one, one-to-many, and many-to-many associations (see docs)

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

A.hasOne(B);    // A HasOne B
A.belongsTo(B); // A BelongsTo B
A.hasMany(B);   // A HasMany B
A.belongsToMany(B, { through: 'C' }); // A BelongsToMany B through the junction
table C
```

DEFINING ASSOCIATIONS

```
import { DataTypes, Model, Sequelize } from "sequelize";

//create connection
const database = new Sequelize("sqlite:mydb.sqlite");

const User = database.define('User', {/*model specs*/});

class Todo extends Model {}

Todo.init({/*model specs*/},{/*options*/});

User.hasMany(Todo);   //Sequelize will define a foreign key Todo -> User
Todo.belongsTo(User);

database.sync({force: true}).then(()=>{ console.log("Database synchronized.") });
```

CREATING LINKED DATA

- By default, Sequelize creates a column named **RefModelRefModelPK**
- In our case, it created a **UserId** column in Todo

```
let riff = await User.create({
  name: "Riff Raff",
  age: 26
});
console.log(`Riff Raff saved to database with id ${riff.id}.`);

let todo = await Todo.create({
  todo: "Do the time warp again",
  done: true,
  UserId: riff.id
});
```

CREATING LINKED DATA

- An instance can also be created along with nested associations in a single step, provided all elements are new

```
let janet = await User.create({
  name: "Janet",
  age: 22,
  Todos: [
    {todo: "Learn Sequelize"},
    {todo: "Learn Express"}
  ]
}, {
  include: [Todo]
});

// code creating Riff Raff
// and its Todo goes here
```

name	age	id	createdAt	updatedAt
Janet	22	1	2023-12-09 10:03:50	2023-12-09 10:03:50
Riff Raff	26	2	2023-12-09 10:03:50	2023-12-09 10:03:50

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	0	1	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Learn Express	0	2	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

RETRIEVING DATA

- Sequelize models provide simple ways to retrieve data
- See the docs for a complete reference on queries

```
let todos = await Todo.findAll();
for(let i of todos)
  console.log(`${i.id}: ${i.todo}`);
```

```
SELECT *
FROM Todos
```

```
import { Op } from "sequelize";

let todos = await Todo.findAll({
  where: {
    id:   { [Op.gte] : 2 },
    todo: { [Op.like]: '%Express%’ }
  }
});
```

```
SELECT *
FROM Todos
WHERE id >= 2 AND todo LIKE '%Express'
```

RETRIEVING DATA

- By default, associations are not loaded when we retrieve data
- We can specify we want to load some associations as well

```
let todos = await Todo.findAll({include: [User]});
for(let item of todos)
  console.log(`${item.todo} - ${item.User?.name}`);
```

- We can retrieve an entitity given its primary key using **.findByPk(pk)**

```
let t = await Todo.findByPk(3);
console.log("t:", t.id, t.todo, t.done, t.UserId);
```

```
Executing: SELECT `todo`, `done`, `id`, `createdAt`, `updatedAt`, `UserId` FROM
`Todos` AS `Todo` WHERE `Todo`.`id` = 3;
t: 3 Do the time warp again false 2
```

UPDATING AND DELETING DATA

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	0	1	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Learn Express	0	2	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

```
let learnSequelize = await Todo.findByPk(1);
learnSequelize.setDataValue('done', true);
await learnSequelize.save(); //updates entity on db

let learnExpress = await Todo.findByPk(2);
await learnExpress.destroy(); //deletes entity from db
```

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	1	1	2023-12-09 10:03:50	2023-12-09 10:15:12	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

ORM: PROS

- **Abstraction**: Work with objects and models instead of SQL
- **Productivity**: less boilerplate code, automated schema gen.
- **Portability**: easy to switch between supported RDBMS
- **Readability** and **consistency**: more readable than raw SQL; enforce specific coding patterns and conventions.

CONS

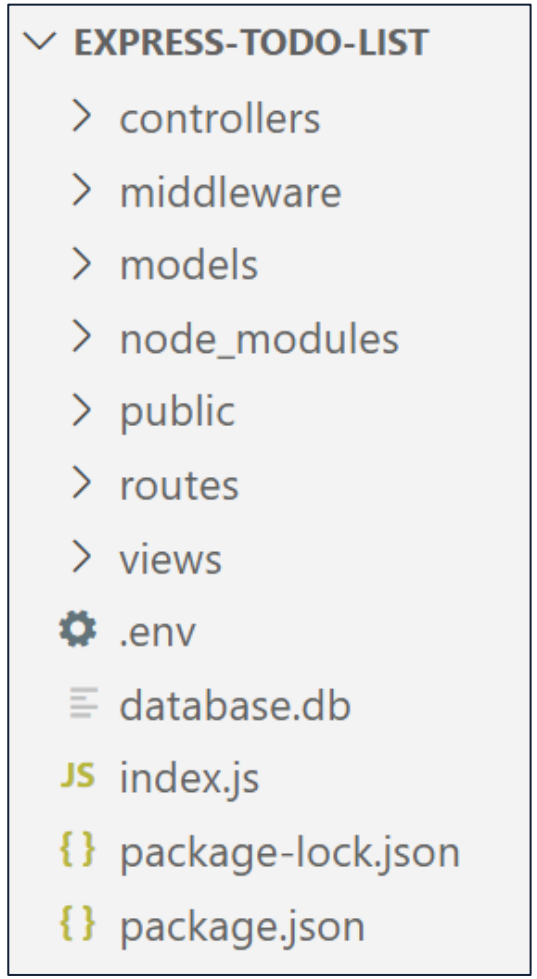
- Steeper learning curve
- **Complexity**: some operations might be more complex with an ORM
- **Vendor Lock-in**: might be difficult to switch to different ORMs
- **Performance**: ORMs might introduce an overhead for some operations, especially when used improperly

STRUCTURE OF AN EXPRESS APP

ORGANIZING AN EXPRESS WEB APP

- Express is unopinionated. As a consequence there is not a single ***right*** way of organizing a web app
- In the Web Technologies course we'll try our hand at an organization
  - It's not carved in stone! Feel free to think about it and improve it!

ORGANIZING AN EXPRESS WEB APP



- **views** contains templates
- **routes** contains definition of Routers. This code is responsible for directing requests to the appropriate controller
- **controllers** contains code implementing business logic for handling requests
- **models** contains definition of data models and database connections
- **middleware** contains custom middleware
- **public** contains static files organized in directories

WEB APP CONFIGURATION

WEB APP CONFIGURATION

Web apps often depend on **sensitive information** and **environment-specific settings**

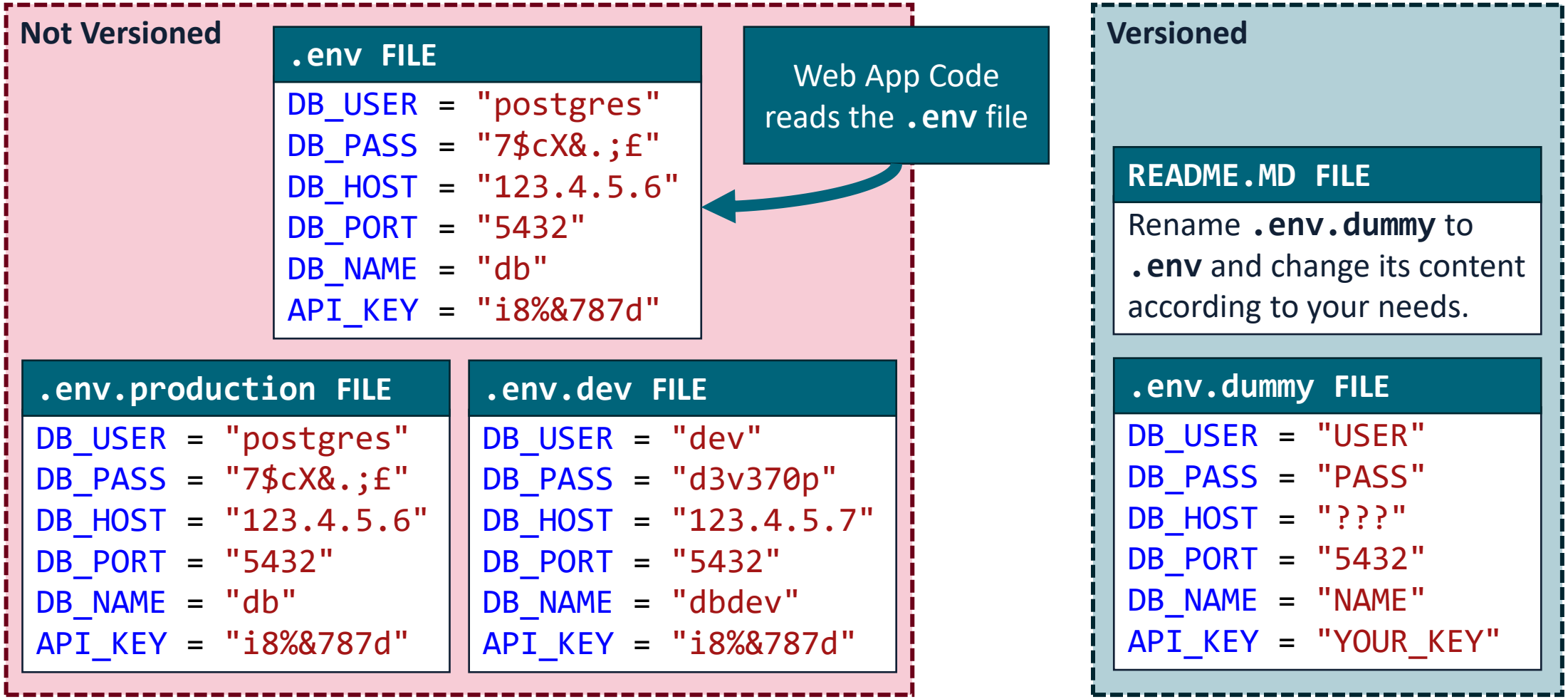
- **Sensitive:** API keys, database passwords, AWS credentials
- **Environment-specific**: URL of the production/development/staging database, logging levels, etc…

- Hard-coding these settings in our code is not a good practice!
  - **Not safe** when we version our code in a public repository
  - Might need to change multiple files to change these settings
  - Leads to different «versions» of the Web App (e.g.: development vs production), which all need to be updated and maintained separately
- These data should be stored in a dedicated **configuration file**

CONFIGURATION FILES

- Configuration files should be the single point where these settings can be changed
- They should not be versioned (i.e.: put ‘em in the **.gitignore** file)
- A dummy version can be versioned instead, so each user can modify it and include its own settings
- One way to manage settings is using a **.env** file in the project root
- Node.js packages such as **dotenv** can be used to read the .env file and make its contents available (e.g.: in the global **process.env** object)

WEB APP CONFIGURATION



EXPRESS TO-DO LIST WEB APP

The same old web app, now with Express (with a few spicy additions!)

TO–DO LIST APP WITH EXPRESS

- Let’s re-implement our To-do List App with Express
- We’ll make this interesting by adding a few more features
  - Users will be able to mark to-do items as «done» or «not done yet»
  - Users will be able to **modify** or **delete** to-do items
  - We’ll use the Sequelize ORM to manage data persistence

LET'S LOOK AT THE CODE

- Source code available in the course materials
- **Will the lecturer make it through this live demo?**
- **Some highlights are reported in the remaining slides**

CONFIGURING MIDDLEWARE (INDEX.JS)

```
const app = express();
const PORT = 3000;

app.set("view engine", "pug"); //use pug as the default template engine

app.use(morgan('dev'));
app.use(express.static("public"));
app.use(express.urlencoded({extended: false}));
app.use(session({
  secret: 'fu-tzu the great master',
  resave: false, saveUninitialized: false,
  cookie: { maxAge: 10*60*1000 /*10 minutes, in ms */ }
}));
app.use(flash());
app.use(exportFlashMessagesToViews);
app.use(exportAuthenticationStatus);
```

CONFIGURING ROUTES (INDEX.JS)

```
//routes
app.use(homepageRouter); app.use(resetRouter);
app.use(authenticationRouter); app.use(todoRouter);

//catch all, if we get here it's a 404
app.get('*', function(req, res){
  res.status(404).render("errorPage", {code: "404", description: “Not found."});
});

//error handler
app.use( (err, req, res, next) => {
  console.log(err.stack);
  res.status(err.status || 500).render("errorPage", {
    code: (err.status || 500), description: (err.message || "An error occurred")
  });
});

app.listen(PORT);
```

ROUTES: TODOROUTER

```
export const todoRouter = new express.Router();

todoRouter.use(enforceAuthentication); //middleware used in this router

todoRouter.get("/todo", (req, res, next) => {
  TodoController.getTodosForCurrentUser(req).then(todoItems => {
    res.locals.todoItems = todoItems;
    res.render("todo");
  }).catch(err => {
    next(err);
  });
});
```

CONTROLLER: TODOCONTROLLER

```
import { Todo } from "../models/Database.js";

export class TodoController {

  static async getTodosForCurrentUser(req){
    return Todo.findAll({
      where: {
        UserUserName: req.session.username
      }
    })
  }

  // other methods
}
```

AUTHENTICATION

```
export class AuthController {

  static async checkCredentials(req, res){
    let user = new User({ //user data specified in the request
      userName: req.body.usr, password: req.body.pwd
    });

    let found = await User.findOne({
      where: { userName: user.userName, password: user.password }
    });

    if(found === null) {
      return false;
    } else { //credentials are valid. We create a session
      req.session.isAuthenticated = true; req.session.username = found.userName;
      return true;
    }
  }
```

AUTHENTICATION MIDDLEWARE

```
/**
 * This middleware ensures that the user is currently authenticated. If not,
 * redirects to login with an error message.
 */
export function enforceAuthentication(req, res, next){
  if(!req?.session?.isAuthenticated){
    req.flash("error", "You must be logged in to access this feature.");
    res.redirect("/auth");
  } else {
    next();
  }
}
```

REFERENCES

- **ORM Hate**
  By Martin Fowler
  Available at https://martinfowler.com/bliki/OrmHate.html and archived here.
- **Getting Started: Sequelize**
  Sequelize Official Docs
  Available at https://sequelize.org/docs/v6/getting-started/