Study Guide / Cheat Sheet: Fullstack Web Development — Week 2

PostgreSQL, Prisma ORM, and Authentication

---

1. PostgreSQL Quick Reference

CREATE TABLE Syntax & Data Types

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE tasks (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    status VARCHAR(20) DEFAULT 'pending' CHECK (status IN ('pending', 'in-progress', 'completed')),
    user_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Common Data Types:

Data Type	Description	Example
`SERIAL`	Auto-incrementing integer (primary key often)	`id SERIAL`
`VARCHAR(n)`	Variable-length string, max `n` characters	`username VARCHAR(50)`
`TEXT`	Variable-length string, unlimited length	`content TEXT`
`INTEGER`	Whole number	`age INTEGER`
`BOOLEAN`	True or false	`is_active BOOLEAN`
`TIMESTAMP`	Date and time	`created_at TIMESTAMP`

Constraints:

- `PRIMARY KEY` : Uniquely identifies each row; implies `NOT NULL` and `UNIQUE`
- `FOREIGN KEY` : Links a column to the primary key of another table
- `NOT NULL` : Column cannot have a `NULL` value
- `UNIQUE` : All values in a column must be distinct
- `DEFAULT` : Sets a default value if none is provided
- `CHECK` : Ensures values satisfy a boolean expression

DML (Data Manipulation Language)

INSERT

```
INSERT INTO users (username, email, password_hash)
VALUES ('johndoe', 'john@example.com', 'hashedpassword123');
```

SELECT

```
SELECT * FROM users;
SELECT title, status FROM tasks WHERE user_id = 1;
```

UPDATE

```
UPDATE users SET email = 'new@example.com' WHERE username = 'johndoe';
```

DELETE

```
DELETE FROM tasks WHERE id = 5;
```

JOINs and Advanced Queries

```
-- INNER JOIN: rows matching in both tables
SELECT u.username, t.title, t.status
FROM users AS u
INNER JOIN tasks AS t ON u.id = t.user_id;

-- LEFT JOIN: all users, even those with no tasks
SELECT u.username, t.title
FROM users AS u
LEFT JOIN tasks AS t ON u.id = t.user_id;

-- ORDER BY
SELECT * FROM tasks ORDER BY created_at DESC;

-- GROUP BY with COUNT
SELECT user_id, COUNT(*) AS total_tasks
FROM tasks
GROUP BY user_id
HAVING COUNT(*) > 2;
```

Common `psql` Commands

Command	Description
`psql -U user_name`	Connect to PostgreSQL
`\l`	List all databases

`\c database_name`	Connect to a database
`\dt`	List tables
`\d table_name`	Describe a table
`\q`	Exit psql

---

2. Prisma ORM Quick Reference

schema.prisma

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id           Int      @id @default(autoincrement())
  email        String   @unique
  username     String   @unique
  passwordHash String
  avatarUrl    String?
  createdAt    DateTime @default(now())
  tasks        Task[]
}

model Task {
  id          Int      @id @default(autoincrement())
  title       String
  description String?
  status      String   @default("pending")
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  user        User     @relation(fields: [userId], references: [id])
  userId      Int
}
```

Field Attributes

Attribute	Purpose
`@id`	Primary key
`@unique`	Unique constraint
`@default(value)`	Default value (`autoincrement()`, `now()`, `uuid()`)

`?`	Optional/nullable field
`@relation(...)`	Define relationships
`@updatedAt`	Auto-update timestamp

Migration Commands

```
npx prisma migrate dev --name init   # Create + apply migration
npx prisma generate                  # Regenerate client (ALWAYS after schema changes!)
npx prisma studio                    # Visual database browser
```

Prisma Client CRUD

```
// Initialize
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

// CREATE
const user = await prisma.user.create({
  data: { username: 'jane', email: 'jane@example.com', passwordHash: 'hashed' }
});

// FIND MANY
const tasks = await prisma.task.findMany({
  where: { userId: 1 },
  include: { user: { select: { id: true, username: true } } },
  orderBy: { createdAt: 'desc' }
});

// FIND UNIQUE
const task = await prisma.task.findUnique({ where: { id: 1 } });

// UPDATE
const updated = await prisma.task.update({
  where: { id: 1 },
  data: { status: 'completed' }
});

// DELETE
await prisma.task.delete({ where: { id: 1 } });
```

Error Handling

```
try {
  await prisma.user.create({ data: { ... } });
} catch (error) {
  if (error.code === 'P2002') {
    // Unique constraint violation
```

```
    res.status(409).json({ message: 'Already exists' });
  }
}
```

---

3. Authentication Quick Reference

bcrypt — Password Hashing

```
const bcrypt = require('bcrypt');

// Hash a password (registration)
const hashedPassword = await bcrypt.hash('userPassword', 10);

// Compare password (login)
const isMatch = await bcrypt.compare('userPassword', hashedPassword); // true/false
```

JWT — JSON Web Tokens

```
const jwt = require('jsonwebtoken');

// Generate token (after login/register)
const token = jwt.sign(
  { userId: user.id },
  process.env.JWT_SECRET,
  { expiresIn: '24h' }
);

// Verify token (in middleware)
const decoded = jwt.verify(token, process.env.JWT_SECRET);
// decoded.userId === user.id
```

Token Structure: `header.payload.signature`

1. **Header**: `{ "alg": "HS256", "typ": "JWT" }`
2. **Payload**: `{ "userId": 123, "iat": 1678886400, "exp": 1678890000 }`
3. **Signature**: HMAC-SHA256(base64(header) + "." + base64(payload), secret)

Registration Flow

```
async function register(req, res) {
  const { username, email, password } = req.body;
  // 1. Validate input
  if (!username || !email || !password) return res.status(400).json({ message: 'All fields required' });
  // 2. Hash password
  const passwordHash = await bcrypt.hash(password, 10);
  // 3. Create user in database
  const user = await prisma.user.create({ data: { username, email, passwordHash } });
  // 4. Generate JWT
  const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET, { expiresIn: '24h' });
```

```
  // 5. Return token
  res.status(201).json({ token, user: { id: user.id, username, email } });
}
```

Login Flow

```
async function login(req, res) {
  const { email, password } = req.body;
  // 1. Find user by email
  const user = await prisma.user.findUnique({ where: { email } });
  if (!user) return res.status(401).json({ message: 'Invalid credentials' });
  // 2. Compare password
  const isMatch = await bcrypt.compare(password, user.passwordHash);
  if (!isMatch) return res.status(401).json({ message: 'Invalid credentials' });
  // 3. Generate JWT
  const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET, { expiresIn: '24h' });
  // 4. Return token
  res.status(200).json({ token, user: { id: user.id, username: user.username, email } });
}
```

Auth Middleware

```
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // "Bearer <token>"
  if (!token) return res.status(401).json({ message: 'Token required' });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.userId;
    next();
  } catch (error) {
    return res.status(401).json({ message: 'Invalid or expired token' });
  }
}

// Usage: router.get('/tasks', authenticateToken, getTasksHandler);
```

---

4. Common Mistakes to Avoid

1. **SQL Injection** — Never concatenate user input into queries. Use Prisma or parameterized queries.
2. **Storing plain text passwords** — Always hash with bcrypt before storing.
3. **Exposing password_hash in responses** — Use `select` to choose safe fields only.
4. **Hardcoding secrets** — Use `.env` files with `dotenv`. Never commit `.env` to git.
5. **Forgetting `npx prisma generate`** — Run after every schema.prisma change.
6. **Not handling Prisma errors** — Wrap operations in try/catch, check error codes.
7. **Same error message for "user not found" and "wrong password"** — Use generic "Invalid credentials" to prevent user enumeration.
8. **Forgetting `await`** — All Prisma and bcrypt operations are async.

9. **Not validating JWT_SECRET exists** — App should fail fast if secret is missing.
10. **Returning the full user object** — Always select specific safe fields.