Node.js Orientation

Amr A Khllaf

July 30, 2025

Contents

1 Application Types

- **Web Applications**
- **Mobile Applications**
- **Desktop Applications**

1.1 Electron (Desktop Applications)

- Framework for building cross-platform desktop apps
- **Leverages web technologies (HTML, CSS, JavaScript)**
- Built on Node.js and Chromium
- Provides access to native OS features
  Electron Documentation

1.2 React Native (Mobile Applications)

- Framework for building native mobile apps using React
- **Uses JavaScript and React components**
- Compiles to native code for iOS and Android
- Enables access to native mobile features
  React Native Documentation

1.3 Node.js (Web Applications)

- JavaScript runtime built on Chrome's V8 engine
- **Enables server-side development with JavaScript**
- Handles HTTP requests, databases, and more
- Can be used with frameworks like Express.js
  Node.js Documentation

1.4 Summary

- **Electron**: For desktop apps using web technologies.
- **React Native**: For mobile apps using React.
- **Node.js**: For web applications using JavaScript.
- All three allow developers to use JavaScript across platforms.
- Each framework provides access to platform-specific features while maintaining a unified codebase.
- Choose based on your target platform and application requirements.
- Explore the documentation for each framework to get started.

---

2 Software Development Life Cycle (SDLC)

2.1 Agile Project Management with SCRUM

- **SCRUM** is an Agile framework emphasizing iterative progress, collaboration, and adaptability.
- Promotes frequent delivery of small, functional increments.
- Encourages open communication and teamwork.
- Key roles:
  - **Scrum Master**: Facilitates the process and removes obstacles.
  - **Product Owner**: Defines product vision and prioritizes requirements.

- **Development Team**: Delivers the product incrementally.

2.2 SDLC Phases

1. **Planning**: Define project goals, scope, and resources.
2. **Analysis**: Gather and document requirements, evaluate user needs, and assess feasibility.
3. **Design**: Develop system architecture and detailed specifications.
4. **Implementation**: Code, integrate, and test the solution.
5. **Deployment**: Release the product to end users.
6. **Maintenance**: Provide support, updates, and enhancements.

2.3 Web Application Development

- A web application runs on a web server and is accessed via a browser.
- Typically uses a client-server architecture: the client is the browser, the server hosts application logic and data.
- **Analysis**: Identify user needs and system requirements.
- **Frontend**: Design user interfaces and user experience.
- **Backend**: Develop server-side logic, APIs, and database interactions.
- **Testing**: Ensure functionality, performance, and security.

2.3.1 Requirements Gathering & Analysis

- **Stakeholder Engagement**: Conduct interviews and workshops to collect requirements.
- **Use Case Development**: Define scenarios illustrating user interactions.
- **User Stories**: Write concise, user-focused feature descriptions.
- **Prototyping**: Create mockups or wireframes for early feedback.
- **Software Requirements Specification (SRS)**: Compile all requirements and acceptance criteria into a structured document.
- **Validation and Approval**: Review requirements with stakeholders for accuracy and consensus.

2.3.2 Frontend Development

1. **UX Design**: Focus on intuitive navigation and accessibility.
   - Visualize SRS with wireframes and prototypes.
2. **UI Design**: Create visually appealing, brand-aligned interfaces.
3. **UI Development**: Implement interfaces using HTML, CSS, and JavaScript frameworks (e.g., React, Angular, Vue.js).
4. **Frontend Frameworks**: Use frameworks like **React, Angular, or Vue.js** for dynamic, responsive UIs.

2.3.3 Backend Development

1. **Programming Languages**: Use JavaScript (Node.js), Python, Ruby, or Java for server-side logic.
2. **Database Integration**: Connect to SQL (MySQL) or NoSQL (MongoDB) databases.

3. **Frameworks**: Employ **Express.js (Node.js)**, **Django (Python)**, or **Ruby on Rails** for backend development.
4. **API Development**: Create RESTful or GraphQL APIs for frontend-backend communication.

2.4 SDLC Models

- **Waterfall Model**: Linear, sequential approach; best for well-defined requirements.
- **Agile Model**: Iterative, incremental, and flexible; ideal for evolving requirements.
- **Spiral Model**: Combines iterative development with risk assessment and prototyping; suited for large, complex projects.

2.5 Importance of SDLC

- Ensures a systematic, organized development process.
- Reduces risks and manages complexity.
- Improves product quality and alignment with user needs.
- Facilitates better planning, tracking, and communication.

---

3 MVC & MC Architectural Patterns

- Architectural patterns structure applications for maintainability and scalability.

3.1 MVC (Model-View-Controller)

- **Model**: Manages data and business logic.
- **View**: Handles the user interface and presentation.
- **Controller**: Processes user input, updates the model, and selects the view.

Key Points:

- Promotes separation of concerns between data, UI, and control logic.
- Common in frameworks like Express.js, Django, and Ruby on Rails.

3.2 MC (Model-Component)

- **Model**: Handles data and business logic.
- **Component**: Encapsulates UI and behavior into reusable units.

Key Points:

- Focuses on modularity and reusability.
- Widely used in modern frontend frameworks (React, Angular, Vue.js).

3.3 Comparison

- **MVC**: Separates data, UI, and control logic.
- **MC**: Prioritizes modular, reusable components.

- **Usage**: MVC is common in traditional web apps; MC is foundational in component-based frontend development.

---

4 Testing

Testing ensures software quality and reliability through both process-oriented and product-oriented activities.

4.1 Quality Assurance (QA)

- **Definition:** Proactive process to establish and maintain procedures that prevent defects.
- **Activities:**
  - Develop and enforce test plans, standards, and best practices.
  - Conduct process audits and reviews.
  - Provide training and continuous improvement.

4.2 Quality Control (QC)

- **Definition:** Reactive process to evaluate the product and identify defects.
- **Activities:**
  - Design and execute test cases.
  - Perform unit, integration, system, and acceptance testing.
  - Document and track defects.

Summary:

- **QA** ensures the right processes for building quality software.
- **QC** verifies the final product meets requirements.
- Both are essential for robust, reliable, user-friendly applications.

---

5 Staging Environment & User Acceptance Testing (UAT)

5.1 Staging Environment

A **staging environment** is a pre-production setup mirroring production, used for final testing before deployment.

Purpose:

- Identify and resolve issues not found in earlier stages.

Benefits:

- Safe space to test features and updates.
- Ensures stability and performance under production-like conditions.
- Minimizes risk of bugs reaching end users.

Best Practices:

- Maintain close parity with production (infrastructure, configurations, data).
- Regularly update staging with latest code and representative data.
- Use automated deployment pipelines for consistency.

5.2 User Acceptance Testing (UAT)

UAT is the final testing phase where end users validate the application against their requirements.

Purpose:

- Confirm the software fulfills business needs and is ready for release.

Process:

- Engage users to execute real-world scenarios.
- Gather feedback and address issues.
- Obtain formal sign-off before deployment.

Benefits:

- Ensures user-friendliness and business alignment.
- Reduces post-release issues by validating with real users.

---

6 Node.js & Core Modules

- Node.js is a JavaScript runtime built on Chrome's V8 engine for scalable network applications.
- Enables server-side programming with JavaScript, supporting full-stack development.
- Event-driven and non-blocking, ideal for I/O-intensive apps.
- Rich ecosystem of libraries and frameworks (e.g., Express.js).
- Supports asynchronous programming for efficient request handling.
- Used for RESTful APIs, real-time apps, and microservices.
- Easily deployable on platforms like AWS, Azure, and Heroku.
- NPM provides access to a vast repository of open-source libraries.
- Strong community and extensive documentation.
- Suitable for high-concurrency applications like chat, gaming, and collaboration tools.