**Principles**

* **Agile development** *(Methodology - project management)* **-** several iterative and incremental software development methodologies. A way of working as a team.
  + Methodology
  + Iterative cycle - code for a while, check the results, organize what's needed, and work again. Embrace changes revisit the plan constantly.
  + Streamlined i.e. moving fast (faster and easier work methods) - favor coding over too much planning and tons of documentation. A lot of quick meetings.
  + Time-boxed - plan work by time not features. Plan timed work.
  + Very collaborative - all talk to each other constantly. And the client needs to be responsive.
* Waterfall technique - in this methodology, a specification of the app is written, it’s signed by the client, and a blueprint is build from it, and once it's done no changes are made.
* **Inversion of Control (IoC) -** Frameworks are different from libraries in that they have the concept of Inversion of control.
  + In case of libraries, your code relies on them for specific tasks, but it makes most of the work. So the code is executes, gives control to the library for something specific, then resumes control.
  + In case of a framework - Makes most of the work, where it has control and when it has time it calls your code to do something, and then your code returns control back to the framework.
* **Single Responsibility Principle** - Each class / module should focus on a single task.
* **Open/Closed Principle** - software elements should be open to extension but closed for modification. I.e. classes could be extended but after extension it is no longer needed to be changed.
* **Liskov substitution principle** - In OOP, Ability to replace any instance of a parent class with an instance of child class without any side effects.
* **Dependency inversion principle** - High level objects should not depend of low level implementations. I.e. it should delegate responsibility to lower level components and not deal with implementation of every specific task itself. Basically code should depend upon abstractions, decoupling implementations from each other. And even further, the usage of interfaces between the class and dependency allows for usage of different dependencies interchangeably as long as they implement the same interface as the one required by the class.
* Concern - a part of the system divided on the basis of functionality.
* Core concern - Primary functionality of the system.
* Crosscutting متعددة القطاعات concerns / System-wide concerns:
  + Runtime checking of invariants.
  + Tracing executions.
  + Serializing.
  + Database transaction.
  + Security.
  + Performance enhancement.
  + Notification e.g. email notifications.
  + Error handling.
  + Logging.

**Testing**

Topics:

* Basic testing strategies - structure, dates, etc.
* Mocking, stubbing, asynchrony.

Tools:

* Sinon - Mocking, stub (mocking of modules).
* Mocha.js - Testing framework. (co-mocha is a generator function supporting version)
* Jasmine.js - another option instead of mocha which can replace mocha & sinon.
* Moment.js
* Assert (in some other modules is called expect) - build-in node library - it is not only for testing, for validation of arguments for example.

Additional tools:

* Node’s core assertion library OR:
  + Chai - BDD / TDD assertion library. Allows to write assertions in a more verbose way (should syntax) and also expectations.
  + ShouldJS - BDD style assertions. (close to Chai)
* CoffeeScript - syntactic sugar transpires to javascript.

Testing is a vague word - it can mean different things.

* Beta testing / user acceptance testing - releasing a nearly finished application of a program to let users use it and check what problems may arise.
* Performance testing - profiling tools to measure if we are getting acceptable response time. Is the application running as fast as it should be.
  + Stress testing - one kind of performance testing to check how the program is working under heavy load.
* Integration testing - testing one app integrating with other systems.
* Regression testing - check if software still performs well after making changes. Or interacting with other systems.
* Users’ Usability testing
* Functional testing - testing in the user’s point of view. Basically checking if the code works by using the application like a user would do.
* Unit testing - testing as programmers not users. The smallest pieces possible of the code.
* Quality assurance testing ..

Assert - to state something positively or to be true. Stating things as facts.

Test - a program that check another program (sections of code) works normally. Validating an expectation about something.

* Unit test - tests small feature or isolated functionality of the app.
* Integration tests - testing multiple features working together, or systems interacting with each other.

Testing strategies:

* Test-first: in which test is written first with expected behavior and then the code is written to make it pass.
  + Behavioral driven development - is an evolution of TDD. is more related to the way tests are written, in a manner which is readable, describing the behavior - scenarios/cases and expectations.
  + Test driven development - let the tests drive the development. “Red, Green, refactor” Make code fail, make it run, make it right, then repeat it.

Test double: replace production object for testing purposes.

* Mocks - mocking api calls. Mock object often contains assertions that expect values, in a way that a fake object sometimes does not.
  + Real object that hasn’t been written yet.
  + What you’re calling has a UI / needs human interaction.
  + Slow or difficult to set up.
  + External resource: file system, database, network, printer…
  + Non-deterministic behavior - data that may change from moment to moment or may not always be available.
* Stubs - mock a function response.
* Fakes - delegate a function to a custom implementation. Match the intended method implementation, so they return pre-arranged results. While mocks also verify that any interaction with it is correct (verify interaction).
* Spies - spy on functions and know when they are called and how many times ...

Test code could be arranged into:

* Structures
* Assertions
* Doubles
* Bin folder is an executing directory. Where modules for command line are usually stored.

Mocha could be run locally not globally using “node\_modules/mocha/bin/mocha.

Mocha looks for “\*spec.js” in the name of test files.

* Test specification or also called suite. A spec for short is a container of test, a way to identify the specific feature we are testing.
* Testing a feature with different scenarios with different behaviors (using “it”).
* Node’s core Util library - has an “inherit()” method that mimics ES6 class extends functionality.
* Event path/chain pattern. Chain events on a sequence.
* ORMs:
  + (node-orm2) - Object-relational mapping allows manipulating data from database using object relational paradigm also achieving persistence across different database technologies.
  + Sequelize - promise-based ORM.
  + Waterline ORM - new approach with unified API across databases.
  + Bookshelfs ORM - promise based, Knex SQL query, backbone. (*generally unneeded abstraction for SQL, limits the usage*).
  + MassiveJS - data access tool SQL.
* Sinon
  + Sinon.spy() functionality to watch for asynchronous behaviors.
  + Sinon.stub() - allows to mock functions on objects.
  + Sinon.unwrap() - to restore a stubbed object .
  + Sinon.mock() - spy, stub, expectations together.
* Karma - visual / borwser test runner
* Nock - can intercept an HTTP API call. Can replace Sinon. (super-test another module for testing web apis)
* Underscore clone() method.
* Nedb - Embedded datastore for nodejs. Like sqlite. Can be used instead of mocking databases.
* PhantomJS - headless WebKit
* There are tools for test coverage. Which allows to see how much of the code is being used and how much is tested.
  + Test coverage - how much of the application code is hit by unit tests.
* Can test code that uses timers - in a way that redefines the native clock in the test.
* Assertions are only for developers not for production code. Some tools strip out the tests if they are written in same file.
* What to test and how much ?
  + “test until fear/concern turns to boredom” - We are trying to make sure that every significant path is tested at least once. Written all the tests we wish we had.
  + A test for every branch - if / else / and / or / case / for / while / polymorphism.
  + Action types to test:
    - Initialization
    - Methods
    - Everts
* 3 A rules - to arrange (to get something setup), to act (change it, do something to it), to assert (Check the result, check it work). Each test can be summarized by these three rules.
* Test for expected exceptions / thrown errors.
* Test fixture (تجهيزات) - means that in the tests there are also setup (before) and tear down (after) associated steps.
* We should test our own logic not the language itself. So it's fine to assume would just work.
* Test suite - combining multiple test fixtures / test cases / test classes.
* <https://npmcompare.com/compare/expect,nock,sinon,supertest>
* ORM - Object-relational mapping is a design pattern of accessing a relational database from an object oriented language. Allows interaction with database using instance setter/getters and more. <http://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html> <https://stackoverflow.com/questions/438166/should-i-reflect-database-structure-in-application>