01. Programming paradigms

Programming paradigms are designed to classify programming languages according to their features. Common programming paradigms: 1) **imperative** – the programmer instructs the machine how to change its state (*procedural* which groups instructions into procedures; *object-oriented* which groups instructions with the part of the state they operate on); 2) **declarative** – programmer merely declares properties of the desired result, but not how to compute it (*functional* in which the desired result is declared as the value of a series of function applications; *logic* in which the desired result is declared as the answer to a question about a system of facts and rules; *reactive* in which the desired result is declared with data streams and the propagation of change).

Languages that fall into the imperative paradigm have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit. Meanwhile, in OOP, code is organized into objects that contain a state that is only modified by the code that is part of the object. Most object-oriented languages are also imperative languages. In contrast, languages that fit the declarative paradigm do not state the order in which to execute operations. Instead, they supply a number of available operations in the system, along with the conditions under which each is allowed to execute. The implementation of the language's execution model tracks which operations are free to execute and chooses the order independently.

Object-oriented programming principles

1) Encapsulation means forming a protective barrier around the information contained within a class from the rest of the code. In OOP, we encapsulate by binding the data and functions which operate on that data into a single class. By doing so, private details of class are hidden from the outside and only expose functionality that is important for interfacing with it.

2) Abstraction means abstracting away the implementation details of a class and only presenting a clean and easy-to-use interface via the class' member functions.

3) Inheritance is a mechanism of basing an object/class upon another object/class, retaining a similar implementation. Classes can be organized into hierarchies, where a class might have one or more parent or child classes. That is to say, the child class "IS-A" type of the parent class. Types of inheritance: single, multilevel, hierarchical, multiple, hybrid.

4) Polymorphism is: a) a feature that allows a specific routine to use variables of different types at different times; b) an ability to present the same interface for several different underlying data types; c) an ability of different objects to respond in a unique way to the same message. There are two types of polymorphism: static (compile-time, overloading) and dynamic (dynamic, run time, overriding).

5) Inversion of control is a principle in which custom-written portions of a program receive the flow of control from a generic framework. In traditional programming, the custom code calls into reusable libraries to take care of generic tasks. With IoC, it's the framework that calls into the custom code. The term is related to, but different from, the dependency inversion principle, which concerns itself with decoupling dependencies between high-level and low-level layers through shared abstractions. IoC serves the following purposes:

1. Decoupling the task's execution from implementation
2. Focusing a module on the task it is designed for.
3. Providing freedom to modules from assumptions about how other systems do what they do and instead rely on contracts.
4. Preventing side effects when replacing a module.

Dependency injection is a technique in which an object receives other objects that it depends on, called dependencies. DI involves four roles: **the service objects**, which contain useful functionality; **the**

interfaces by which those services are known to other parts of code; **the client object**, whose behavior depends on using services; **the injector**, which constructs the services and injects them into the client. Types of DI: constructor injection, setter injection, method injection.

Functional programming

1. Pure functions, side effects. Pure function is a function which always returns the same output for the same inputs and has no side-effects (they are produced during an execution of some function which is not reflected in its output and usually not its direct goal). The purity simplifies a testing process, allows chaining of function calls, allows cashing because of referential transparency (property of expression to stay the same after replacing it with its corresponding value and vice-versa).
2. Immutability is a property of object state being unable to change. *Weak immutability* means that only the object's shape (but not its appearance) can't be modified. *Strong immutability* means the impossibility of changing neither keys nor values. More details can be found here: Property flags and descriptors
3. Functions as first-class entities: they refer to first-class citizens. Functions may be treated as any other regular programming objects: may be stored in variables, passed as an argument to a function, returned from a function.
4. Function composition is the process of combining two or more functions in order to produce a new function or perform some computation. Composing functions together is like snapping together a series of pipes for our data to flow through. Put simply, a composition of functions f and g evaluates from the inside out – right to left. There are traditional and compose/pipe approaches (Function composition in JavaScript).
5. High order function is any function which takes a function as an argument, returns a function, or both. Higher order functions are often used to:
    - abstract or isolate actions, effects, or async flow control using callback functions, promises, monads, etc;
    - create utilities which can act on a wide variety of data types;
    - partially apply a function to its arguments or create a curried function for the purpose of reuse or function composition;
    - take a list of functions and return some composition of those input functions.
6. Recursion is the process a function goes through when one of the function's steps involves invoking the function itself. Recursion uses system stack (LIFO approach) to accomplish the task. **Pros**: reduces side effects, makes code more concise and easier to reason about, reduces system resource usage and performs better than the traditional for loop. **Cons**: can lead to stack overflow, more complicated to set up than a traditional for loop.
7. Currying is a process of a multiple arguments function's representation as a chain of functions of one argument. Partial application is a process of creating a function from a curried function by passing an argument to it and storing the result in a variable. What is in common: both currying and partial application functions are not the same functions as the original. They are newly returned functions and take fewer parameters. What is not in common: currying takes only one parameter, unlike partial application which can take more than one. Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. It's really useful in conjunction with recursion because recursive function at second and further calls uses only memoized results. In order to memoize a function, it should be pure so that return values are the same for the same inputs every time. Memoizing is a trade-off between added space and added speed and thus only significant for functions having a limited input range so that cached values can be made more frequently.

Functional programming vs object-oriented programming, composition over inheritance.
Functional: function is the primary unit; pure functions don't have side-effects, follows a more declarative programming model; there are only immutable objects; production of outputs with identical inputs (extremely operational, practical); compiler optimizations and caching.
Object-oriented: objects are the main unit; methods may have side effects; follows an imperative programming approach; can support mutable and immutable objects; stores data in objects, the data is prioritized over the operations; less memory consumption.
Cons of inheritance:

- Tight coupling problem: because child classes are dependent on the implementation of the parent class, class inheritance is the tightest coupling available in object-oriented design;
- Fragile base class problem: due to tight coupling, changes to the base class can potentially break a large number of descendant classes (potentially in code managed by third parties);
- Inflexible hierarchy problem: with single ancestor taxonomies, given enough time and evolution, all class taxonomies are eventually wrong for new use-cases;
- The duplication by necessity problem: due to inflexible hierarchies, new use cases are often implemented by duplication, rather than extension, leading to similar classes which are unexpectedly divergent (it's not obvious which class new classes should descend from).

Reactive programming is like functional programming where functions and data are combined with time and called streams. A stream can be created from anything: array, primitive value, etc. We can merge streams, composite them, have a high order stream which takes another stream as an input or gives as a result. A stream is a sequence of ongoing events ordered in time. Streams can emit three things: a value, an error, or a "completed" signal. To capture these emits, you subscribe to a stream passing a function called observers (Observer design pattern). On top of that, we have a bunch of functions to help us handle streams with all the power of functionality. RxJS is the most common example of RP in JavaScript. Pros: streams, merging, writing declarative code, avoiding callback hell, threading and asynchronous mechanisms implementation, purity, avoiding implementation details with a focus on business goals. Cons: only hard debugging, making documentation, memory consumption, time to start, managing concurrency, data immutability required, complexity of testing, learning curve.

02. Communication protocols

Open Systems Interconnection model

Layer	Data Unit	Function	Examples	Equipment
Application	Data	High-level APIs, including resource sharing, remote file access	HTTP, FTP, POP3, Telnet, WebSocket, SMTP, DHCP	Hosts, firewall
Presentation		Translation of data between service and application (i.e., character encoding, data compression, encrypt / decrypt)	ASCII, JPEG, EBCDIC, MIDI, MPEG	
Session		Managing communication sessions (i.e., continuous exchange of information with back-and-forth transmissions)	RPC, PAP, L2TP, gRPC, SDP, SMPP	
Transport	Segment, datagram	Reliable transmission of data between points on a network (i.e., segmentation, acknowledgement, multiplexing)	TCP, UDP, RDP, SCTP	

Network	Packet	Structuring and managing a multi-node network (i.e., addressing, routing and traffic control)	IPv4, IPv6, RIP, OSPF, EIGRP, ICMP	Router, gateway, firewall
Data link	Frame	Reliable transmission of data frames between nodes which are connected at a physical layer	IEEE 802.11, Ethernet, ARP, PPPoE	Bridge, switch, access point
Physical	Bit, symbol	Transmission and reception of raw bit streams over a physical medium	IEEE 802.11, USB, RJ	Hub, repeater

ARP (Address Resolution Protocol - Wikipedia) is used to resolve network layer addresses to data link layer addresses in some LAN. Most often it's IP address to MAC (media access control) address. When a host needs to talk to a host with a given IP address, it references the ARP cache to resolve the IP address to a MAC address. If the address is not known, a request is made asking for the MAC address of the device with the IP address. ARP is a request-response protocol whose messages are encapsulated by a link-layer protocol. It is communicated within the boundaries of a single network, never routed across internetworking nodes. This property places ARP into the Link layer. Inverse Address Resolution Protocol (Inverse ARP or InARP) is used to obtain Layer 3 address by Layer 2 address. **ARP spoofing** – look further.

IP (Internet Protocol):

- Internet Protocol - Wikipedia
- https://github.com/alexanderteplov/computer-science/wiki/IP

TCP (Transmission Control Protocol), UDP (User Datagram Protocol):

- Transmission Control Protocol - Wikipedia
- User Datagram Protocol - Wikipedia
- https://github.com/alexanderteplov/computer-science/wiki/TCP-and-UDP

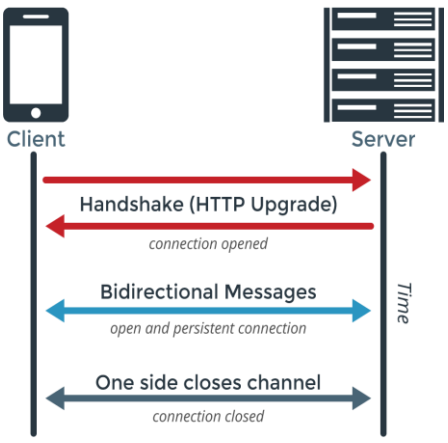
HTTP (Hypertext Transfer Protocol):

- An overview of HTTP
- Transport Layer Security (TLS) - High Performance Browser Networking (O'Reilly)
- Hypertext Transfer Protocol - Wikipedia

IPC (inter-process communication), RPC (remote procedure call), JSON-RPC, gRPC:

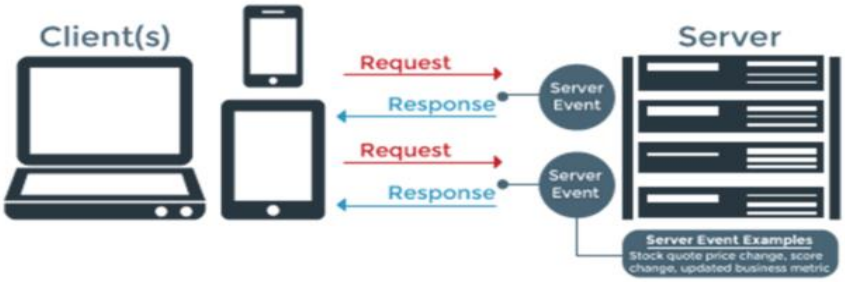
- Inter-process communication - Wikipedia
- Remote procedure call - Wikipedia
- JSON-RPC - Wikipedia
- gRPC - Wikipedia

WebSockets vs Polling



WebSocket is a computer communication protocol that provides full-duplex communication channels over a single TCP connection. It's compatible with TCP (4th level OSI protocol), 7th level OSI protocol, works over 80 and 443 (in case of TLS encrypted) ports, supports HTTP proxies and intermediaries. To achieve compatibility, the WebSocket handshake uses an upgrade header to update the protocol to the WebSocket protocol. It enables client-server interaction with lesser overheads, providing real-time data transfer from and to the server. WebSockets keeps the connection open, allowing messages to be passed back and forth between the client and the server (two-way ongoing conversation can take place). Overview: WebSocket.

In AJAX polling, a client makes XHR/Ajax requests to the server repeatedly at some regular interval to check for new data. A client initiates requests at small regular intervals (0.5 sec), the server prepares the response and sends it back to the client just like normal HTTP requests. Making repeated requests to the server wastes resources as each new incoming connection must be established, the HTTP headers must be passed, a query for new data must be performed, and a response (usually with no new data to offer) must be generated and delivered. The connection must be closed and any resources cleaned up.



As in regular polling, rather than having to repeat this process multiple times for every client until new data becomes available, Long polling is a technique where the server elects to hold a client connection open for as long as possible, delivering a response only after data becomes available or timeout threshold has been reached. After receiving a response, the client immediately sends the next request. On the client-side, only a single request to the server needs to be managed. When the response is received, the client can initiate a new request, repeating this process as many times as necessary. Flow for long polling:

1. A client initiates an XHR/AJAX request, requesting some data from a server.
2. The server does not immediately respond with request information but waits until there is new information available.
3. When there is new information available, the server responds with new information.
4. The client receives the new information and immediately sends another request to the server restarting the process.

Challenges in long polling: message ordering and delivery guarantees; message ordering can't be guaranteed if the same client opens multiple connections to the server (thus, possible message loss in case of client's inability to receive it); performance and scaling; device support and fallbacks.

REpresentational State Transfer (REST) is a software architectural style that defines constraints for organizing web systems. Peculiarities of REST: client-server architecture, statelessness, cacheability, layered system, code on demand (optional), uniform interface. Richardson maturity model consists of next levels:

- Level 0 – one URL, one method;
- Level 1 – multiple URLs (resources), one method;
- Level 2 – multiple URLs, multiple methods (HTTP Verbs, HTTP semantic methods);
- Level 3 – all from level 2 + ability API of self-documenting (GET requests can return a list of URLs with possible actions)

CRUD – GET, POST, PUT, DELETE (an approach of usage PUT for creation is also possible).
Idempotent methods: OPTIONS, GET, HEAD, PUT, DELETE.
Safe methods (could be cached by browser, proxy, gateway server): OPTIONS, GET, HEAD.
Contract First. Pros – parallel work; possible code generation (stubs, mocks); higher level of abstraction, less implementation-specific details; better design and easier code reusing. Cons – more effort on start; extra costs to maintain and update a contract.
Code First. Pros – easy to get a contract with generation from code; auto-sync between contract and code with contract generation. Cons – no parallel work; more chaotic contract; higher coupling with implementation details (programming language, platform).
GraphQL is database-agnostic, transport-agnostic, strongly typed (works with statical analyzers. compatible with TypeScript) and a client-server RPC-like architecture:

- Function is defined on a server, client is specified with a request pointing to this function and providing its input (there is its output in response);
- The main difference with RPC is the capability of invoking several nesting functions per one request (reception of data from several requests).

Problems of GraphQL: authorization (no out-of-the-box solution for it; proper implementation is not evident); N+1 problem (with nesting, requests may affect performance by getting information with separate requests to DB); API complexity; bundle size (GraphQL requests are included in the bundle); versioning (no out-of-the-box solution for it).

Components of GraphQL web service

1. Network transport system. HTTPS is the most often and best provided with tools.
2. Network server. Mostly it's an HTTP server, and the most common is Express.
3. GraphQL server. All custom servers are built over the original GraphQL.js Server.
4. GraphQL client (most known and feature-rich clients are Relay and Apollo Client).

Types – the way GraphQL describes any entities as objects in its own type language (scalar, enumeration, lists, union and input types, interfaces, query and mutation message types).

Queries – type of messages from a client to a server. A client specifies fields it wants, their hierarchy, and values (*query { stuff { eggs shirt pizza }}*).

Mutations – type of messages from a client to a server for making changes, mutations (in a DB, cache, or anywhere).

Resolvers – functions matching fields described in a query with someplace and a way to get them (e.g., make an SQL request to DB). Or the same way deal with mutations received from a client.

Schema – the schema links all types and resolvers together, documenting for both developers and GraphQL packages your API.

Good client features: dealing with a network (handling requests); storing data and accessing them; normalizing data keeping in a store (for deduplication and atomic usage); pub/sub mechanism for synchronization data between a store and interface components; immediate applying changes to an interface with restoring previous when failed; generating and validating types.

REST vs GraphQL pros/cons

REST	GraphQL
An architectural style largely viewed as a conventional standard for designing APIs	A query language offering efficiency and flexibility for solving common problems when integrating APIs
Deployed over a set of URLs where each of them exposes a single resource	Deployed over HTTP using a single endpoint that provides full capabilities of exposed service
Uses a server-driven architecture	Uses a client-driven architecture
Uses caching automatically	Lacks automatic caching mechanism
Supports multiple API versions	No API versioning
Supports multiple data formats	JSON representation only
Wide range of options for automated documentation (OpenAPI, API Blueprint etc.)	Only a single tool (GraphiQL) is used predominantly for documentation
Uses HTTP status codes to identify errors easily	Complicates handling of HTTP status codes to identify errors

Pros of REST: mature and proven for decades; handles various types of calls and supports various data formats (plain text, HTML, JSON); decoupling of client and server architectures provides great scalability for expanding applications easily.
Cons of REST: prone to under-fetching or over-fetching of data; multiple round trips of requests required to fetch all the data; no specific, standardized methodology of structuring REST APIs.
Pros of GraphQL: presence of a strong type system, which is expressed as a schema, dramatically reduces the effort required to implement queries (type system stipulates the data structure the server can return); suitable for avoiding data's under-fetching or over-fetching; suitable for parallel API development for front and back ends.
Cons of GraphQL: lacks built-in caching capabilities; since it always returns an HTTP status code of 200, whether the request is successful or not, this may complicate error reporting and API monitoring; lacks extensive adoption and support.

Open API specification is a standard format to define structure and syntax REST APIs. OpenAPI documents are both machine and human-readable, which enables anyone to easily determine how each API works. Engineers can use APIs to plan and design servers, generate code and implement contract testing. Pros: generate accurate documentation; create stub code for API development; build mock servers to prototype the interface; test that API requests and responses match the intended contract. A document written to the OpenAPI specification can use either JSON or YAML to express the API's capabilities. These formats are interchangeable and include the same elements. There are three primary areas in every OpenAPI document:

1. Endpoints (i.e. paths appended to the server URL) and the HTTP methods they support. For each method, any parameters that may or must be included in the request and the response formats for the possible HTTP response codes are specified.
2. Reusable components that can be used across multiple endpoints in the API, such as common request parameters and response formats.
3. Meta information, including the title, version, and description of the API, authentication method, and location of the API servers.

Swagger is a set of tools for working with OpenAPI:

- Swagger Editor (API editor for designing APIs with the OpenAPI Specification);
- Swagger UI (visualize OpenAPI Specification definitions in an interactive UI);
- Swagger Codegen (generate server stubs and client SDKs from OpenAPI Specification definitions).

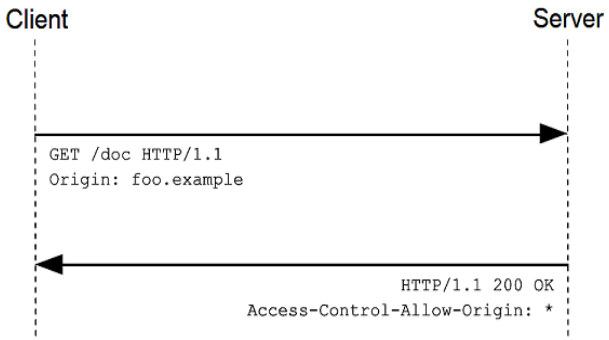
Possibilities: API Design, API Development, API Documentation, API Testing, API Mocking and Virtualization, API Governance, API Monitoring. Limitations: can't be used for documenting APIs with one URL and one method (level 0 from Richardson Maturity Model HTTP), e.g., JSON-RPC.
apiDoc positions itself as *Inline Documentation for RESTful web APIs*, creates documentation from API annotations in your source code. Pros: API web page as structured documentation, supports versioning and comparison between versions, usage of it with JSON-RPC as well. Cons: support only Code first approach, less powerful than Swagger tools.
Postman is a rest client software that started as an chrome extension but is now available as native application only. Postman is basically used for API testing in which APIs can be tested with different types of request method types like post, put etc and parameters, headers and cookies. Apart from setting the query parameters and checking the response, postman also provide different response stats (time, status, headers, cookies etc.) with extra excellent features that can be used with ease.
An environment in Postman is a set of key-value pairs. You can create multiple env in postman which can be switched quickly with a press of a button. There are two types of environment: global and local. They define the scope of the variable to use it in the requests. Most commonly the variable is defined

inside the environment for the same purpose. The most common variable we use is url because url is used in every request and changing it can be very time consuming. When we create an environment inside Postman, we can change the value of the key value pairs and the changes are reflected in our requests. An environment just provides boundaries to variables.
A scope of a variable is defined as the boundaries through which it can be accessed. They are local scope (can be accessed only in the environment in which it was created) and global scope (can be accessed globally in any environment or no environment).

03. Security Basics

Explain what Cross-Origin Resource Sharing (CORS) is and how it works?

The same-origin policy is a security mechanism that generally restricts how a script on some web page can interact with a different web page (another origin; **origin** is a combination of protocol, host and port of URL). It's oriented mostly against embedding legal web pages into criminals' web pages. Legal sites may be totally mocked (e.g., via iframe) or some of their functionality may be borrowed by means of hidden elements (buttons, forms). The same-origin policy doesn't prevent embedding forms, iframes, images, cross-origin scripts. It only will be forbidden by default for any script on a web page to read any cross-origin resources such as iframe (data from it), image (loading to a canvas or background) and to make requests to cross-origin APIs. The protection of something like a malicious image src attribute is not a same-origin policy business, it's up to Content Security Policy.
To say a browser not to limit scripts in particular or any web pages in communications with cross-origin APIs CORS mechanism was introduced. So, CORS is only a relatively safe method to workaround with the same-origin policy. CORS is mostly represented by the group of headers sent by either server or browser.

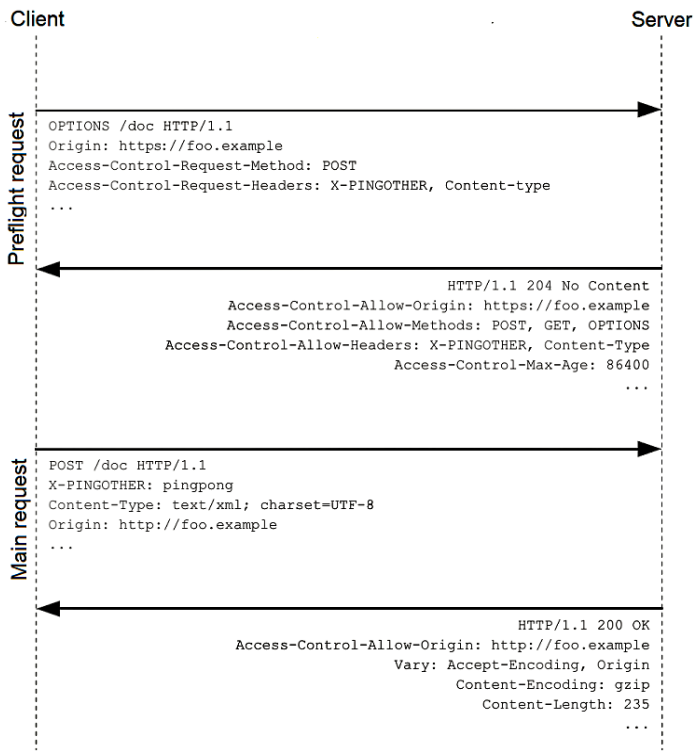


Simple cross-origin requests should meet the following conditions: a) GET, HEAD or POST method; b) allowed headers: Accept, Accept-Language, Content-Language, Content-Type with one of following values: text/plain, application/x-www-form-urlencoded or multipart/form-data; c) no code has called *xhr.upload.addEventListener()* to add an event listener to monitor the upload; d) No ReadableStream object is used in the request.

Preflighted cross-origin requests are any other than simple requests. Browser preflight such a request with an extra request using the OPTIONS method. From a response to it the browser gets to know the CORS settings of the cross-origin server.

Server-side: it's always a server that sets up and controls CORS. Browser only automatically checks CORS settings. The HTTP response headers:

- **Access-Control-Allow-Origin**: <origin> | *;
- **Access-Control-Expose-Headers**: <header-name>[, <header-name>]* (adds the specified headers to the allowlist that JavaScript in browsers is allowed to access);
- **Access-Control-Max-Age**: <delta-seconds> (how long preflight request's results are cached);
- **Access-Control-Allow-Credentials**: true;
- **Access-Control-Allow-Methods**: <method>[, <method>]*;
- **Access-Control-Allow-Headers**: <header-name>[, <header-name>]*.
- The HTTP request headers:
- **Origin**: <origin> | null;
- **Access-Control-Request-Method**: <method>;
- **Access-Control-Request-Headers**: <field-name>[, <field-name>]*.



Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross-Site Scripting (XSS) and data injection attacks. To enable CSP, you need to configure your web server to return the Content-Security-Policy HTTP header. In addition to restricting the domains from which content can be loaded, the server can specify which protocols are allowed to be used; for example a server can specify that all content must be loaded using HTTPS. A complete data transmission security strategy includes not only enforcing HTTPS for data transfer but also marking all cookies with the secure attribute and providing automatic redirects from HTTP pages to their HTTPS counterparts. Sites may also use the Strict-Transport-Security HTTP header to ensure that browsers connect to them only over an encrypted channel.

```
Content-Security-Policy: default-src 'self'; img-src *; media-src
    media1.com media2.com; script-src userscripts.example.com
```

To ease deployment, CSP can be deployed in report-only mode. The policy is not enforced, but any violations are reported to a provided URI. A report-only header can be used to test a future revision to a policy without actually deploying it. You can use the Content-Security-Policy-Report-Only HTTP header to specify your policy. By default, violation reports aren't sent. To enable violation reporting, the report-uri policy directive should be provided with at least one URI to which to deliver reports. If both a Content-Security-Policy-Report-Only header and a Content-Security-Policy header are present in the same response, both policies are honored.

```
Content-Security-Policy-Report-Only: policy
Content-Security-Policy: default-src 'self'; report-uri
    http://reportcollector.example.com/collector.cgi
```

Here are main classes of CSP directives:

1. **Fetch directives** control the locations from which certain resource types may be loaded (default-src, font-src, img-src, media-src, script-src, style-src, etc.).
2. **Document directives** govern the properties of a document or worker environment to which a policy applies (base-uri, sandbox).
3. **Navigation directives** govern to which locations a user can navigate or submit a form (form-action, frame-ancestors, navigate-to).

4. **Reporting directives** control the reporting process of CSP violations (report-to, report-uri).
5. **Other**: require-sri-for, require-trusted-types-for, trusted-types, upgrade-insecure-requests.

Auth Types:

- https://github.com/alexanderteplov/computer-science/wiki/Auth-Types
- Using HTTP cookies
- OAuth 2.0 authentication vulnerabilities | Web Security Academy

Man-in-the-Middle (MITM) attack is an assault with an interceptor in between two communicating hosts. Here are main types of MITM attacks:

1. **Rogue Access Point.** Devices equipped with wireless cards will often try to auto-connect to the access point that is emitting the strongest signal. Attackers can set up their own wireless access point and trick nearby devices to join its domain.
2. **ARP Spoofing.** Because ARP does not provide methods for authenticating ARP replies on a network, ARP replies can come from systems other than the one with the required Layer 2 address. An ARP proxy is a system that answers the ARP request on behalf of another system for which it will forward traffic, normally as a part of the network's design, such as for dial-up internet service. By contrast, in ARP spoofing the answering system, or spoofer replies to a request for another system's address with the aim of intercepting data bound for that system. A malicious user may use ARP spoofing to perform a man-in-the-middle or denial-of-service attack on other users on the network. Various software exists to both detect and perform ARP spoofing attacks, though ARP itself does not provide any methods of protection from such attacks.
3. **mDNS Spoofing.** Multicast DNS is similar to DNS, but it's done on a local area network (LAN) using a broadcast like ARP. This makes it a perfect target for spoofing attacks. Devices such as TVs, printers, and entertainment systems make use of this protocol since they are typically on trusted networks.
4. **DNS Spoofing.** Similar to the way ARP resolves IP addresses to MAC addresses on a LAN, DNS resolves domain names to IP addresses. When using a DNS spoofing attack, the attacker attempts to introduce corrupt DNS cache information to a host in an attempt to access another host using their domain name.

Attack techniques

Sniffing. Attackers use packet capture tools to inspect packets at a low level. Using specific wireless devices that are allowed to be put into monitoring or promiscuous mode can allow an attacker to see packets that are not intended for it to see, such as packets addressed to other hosts.

Packet Injection. An attacker can also leverage their device's monitoring mode to inject malicious packets into data communication streams. Packet injection usually involves first sniffing to determine how and when to craft and send packets.

Session Hijacking. Most web applications use a login mechanism that generates a temporary session token to use for future requests to avoid requiring the user to type a password on every page. An attacker can sniff sensitive traffic to identify the session token for a user and use it to make requests as the user.

SSL Stripping. The SSL Strip takes advantage of the way most users come to SSL websites. The majority of visitors connect to a website's page that redirects through a 302 redirect, or they arrive on an SSL page via a link from a non-SSL site. If the victim wants, for instance, to buy a product and types the URL www.buyme.com in the address bar, the browser connects to the attacker's machine and waits for a response from the server. In an SSL Strip, the attacker, in turn, forwards the victim's request to the online shop's server and receives the secure HTTPS payment page. At this point, the attacker has complete control over the secure payment page. He downgrades it from HTTPS to HTTP and sends it

back to the victim's browser. From now onward, all the victim's data will be transferred in plain text format, and the attacker will be able to intercept it. How to prevent:

- Strong WEP/WAP Encryption on Access Points (the stronger the encryption implementation, the safer);
- Strong router login credentials (essentially to make sure your default router login is changed);
- Virtual Private Network (even if an attacker happens to get on a network that is shared, he won't be able to decipher the traffic in the VPN);
- Force HTTPS (websites should only use HTTPS; users can install browser plugins to enforce always using HTTPS on requests);
- Public Key Pair Based Authentication. MITM attacks typically involve spoofing something or another. Public key pair based authentication like RSA can be used in various layers of the stack to help ensure whether the things you are communicating with are actually the things you want to be communicating with.

OWASP Top 10 (The Open Web Application Security Project)

1) Injection (SQL, NoSQL, LDAP, OS) occurs when untrusted data is sent to an interpreter as part of a command or query. Filtering and validating untrusted input, escaping special characters are effective protective measures.

2) Broken authentication (incorrect implementation of authentication / session management). To avoid this, use multi-factor authentication, strong passwords, properly store and don't expose credentials, limit, delay and log login attempts, limit session lifetime, use strong session IDs, rotate them on login.

3) Sensitive data exposure revealing such data as financial, healthcare, PII. With such data, attackers may conduct credit card fraud, identity theft, or other crimes. To protect: first, classify data processed, stored or transmitted, second, don't store them unnecessarily, third encrypt them properly either at rest or in transit.

4) XML External Entities (XEE). Many older or poorly configured XML processors evaluate external entity references within XML documents. The protection rule is to avoid XML. If it's not the case keep XML processors up-to-date, disable evaluation of external entities, implement whitelisting server-side input validation, use special tools.

5) Broken Access Control (broken authorization). It's typically wrong implemented, forgotten or incomplete control of authorized user's privileges. Use the following advice: with the only exception of public APIs deny all by default, implement access control mechanisms, restrict access on OS level, log failures, invalidate tokens on a server.

6) Security misconfiguration is commonly a result of insecure default configurations, incomplete or incorrect configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. To avoid simply don't do this shit, automate configuring environments, use different credentials, do not install any unnecessary features, keep all up-to-date, use automated control of configuration integrity.

7) XSS occurs whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. Protective measures: using frameworks that automatically escape XSS by design, escaping untrusted HTTP request data, applying context-sensitive encoding, enabling a CSP. XSS can be:

- **reflected** – a script is injected to URL or HTML form, a malicious link is often gotten from a trusted sender via email or messenger (server is unaware of filtering input);
- **stored** – an application or API stores unsanitized user input that is viewed at a later time by another user or an administrator;
- **DOM** – very similar to reflected, but instead of relying on server unawareness of filtering input, it exploits client-side not validated input.

8) Insecure deserialization often leads to remote code execution or replay attacks, injection attacks, and privilege escalation attacks. The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types.
9) Using components with known vulnerabilities. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts. Avoid weird stuff from untrusted sources which you often do not need and if you can't – keep it up-to-date and use automated tools to validate the sources.
10) Insufficient logging and monitoring. Lack of logging and monitoring allows attackers to do their job. All red flags should be logged in a manageable manner, the monitoring system should be able to react in real-time. It's good to use protection frameworks for such purposes.

Cross-Site Request Forgery (CSRF) is an attack vector aimed specifically at automatically authenticating requests. The Basic authentication, Cookies, and OAuth are vulnerable to this attack. It's typically performed by creating a malicious link (button, form, web page) and inducing the user to interact with this link. This way the user unintentionally requests an app he is already logged in to. Such a request may change credentials or perform a banking transaction. In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the action's nature, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality. Key conditions for CSRF are:

- **Relevant action.** There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
- **Cookie-based session handling (certificate-based, basic authentication).** Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
- **No unpredictable request parameters.** The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess (for example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password). The most robust way to defend against CSRF attacks is to include a CSRF token within relevant requests. Token should be: unpredictable with high entropy, as for session tokens in general; tied to the user's session; strictly validated in every case before relevant action is executed. SameSite cookies is an additional defense that is partially effective against CSRF and can be used in conjunction with CSRF tokens.

04. Performance Optimizations

Critical Rendering Path

- https://github.com/alexanderteplov/computer-science/wiki/Critical-Rendering-Path
- Critical Rendering Path | Web Fundamentals | Google Developers
- Understanding the Critical Rendering Path
- Critical rendering path - Web Performance | MDN

High performant animations, repaint/reflow, layout thrashing

- Window.requestAnimationFrame() - Web APIs | MDN
- window.requestIdleCallback() - Web APIs | MDN
- How to create high-performance CSS animations
- CSS Triggers
- **Repaint** goes through all the elements and determines their visibility, color, outline and other visual style properties, then it updates the relevant parts of the screen. **Reflow** on an element

recomputes the dimensions and position of the element, and it also triggers further reflows on that element's children, ancestors and elements that appear after it in the DOM.

- Understanding Repaint and Reflow in JavaScript | The Startup
- Web Animations API
- What forces layout/reflow. The comprehensive list.
- Avoid Large, Complex Layouts and Layout Thrashing
- Browser works in following order: JS → Render Tree → Layout → Painting → Composite Layers; if within one animation frame we change styles and then ask for them, a browser should do an extra layout. Doing it often (e.g., in a cycle or scroll) leads to layout thrashing. To avoid it, read styles at the beginning of the frame and batch reading/writing of them. Prefer Flexbox and Grid over old CSS flow (to make layout lighter).

Improve loading performance and CRP (lazy loading, priority of sources, gzip, minification / uglification)

- JavaScript Start-up Optimization
- Analyzing Critical Rendering Path Performance
- Measuring the Critical Rendering Path

RAIL (https://web.dev/rail/) is a user-centric performance model that provides a structure for thinking about performance. The model breaks down the user's experience into key actions (for example, tap, scroll, load) and helps you define performance goals for each of them. RAIL stands for four distinct aspects of a web app life cycle: response, animation, idle, and load. Users have different performance expectations for each of these contexts, so performance goals are defined based on the context and UX research on how users perceive delays. Make users the focal point of your performance effort. The table below describes key metrics of how users perceive performance delays:

0 to 16 ms	Users are exceptionally good at tracking motion; they dislike it when animations aren't smooth. They perceive animations as smooth so long as 60 new frames are rendered every second. That's 16 ms per frame, including the time it takes for the browser to paint the new frame to the screen, leaving an app about 10 ms to produce a frame.
0 to 100 ms	Respond to user actions within this time window and users feel like the result is immediate. Any longer, and the connection between action and reaction is broken.
100 to 1000 ms	Within this window, things feel part of a natural and continuous progression of tasks. For most users on the web, loading pages or changing views represents a task.
1000 ms or more	Beyond 1 second, users lose focus on the task they are performing.
10000 ms or more	Beyond 10 seconds, users are frustrated and are likely to abandon tasks. They may or may not come back later.

Apply instant loading with the PRPL pattern:

- Push (or preload) the most important resources.
- Render the initial route as soon as possible.
- Pre-cache remaining assets.
- Lazy load other routes and non-critical assets.

With the same tag `<link rel="preload" \>` at the head of an HTML document we can ask the server either to push resources (if supported) or preload them (if there is no 'push' support or the 'nopush' attribute is specified). We should only push/preload the limited number of critical resources. Because, if all the resources have the highest priority – none of them really has. It's a good idea to load

other resources 'in place' when they are needed or during idle time. It's called lazy loading and is mostly applicable to scripts and images (split your code and push/preload or lazy load parts according to their usage). Extra efforts to render the first page as soon as possible (First Paint) are inlining resources or SSR. The first violates cashing, the second can harm time to Interactive. Going further, we can seek help from service workers. They can take assets directly from the cache rather than the server on repeat visits. In simple cases, it could be easily achieved by means of some libraries.

Core web vitals

- Largest Contentful Paint (LCP) determines how quickly the largest and the most important content on the site loads. Reports the render time of the largest image or text block visible within the viewport relative to when the page first started loading. Sites should strive to have LCP of 2.5 seconds or less.
- First Input Delay (FID) determines how quickly a user can start interacting with the site (click, scroll, etc.). FID measures the time from when a user first interacts with a page (i.e. when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to begin processing event handlers in response to that interaction. Sites should strive to have FID of 0.1 seconds or less.
- Cumulative Layout Shift (CLS) determines how much the layout is shifted and if it's stable or not. A layout shift occurs any time a visible element changes its position from one rendered frame to the next. CLS is a measure of the largest burst of layout shift scores for every unexpected layout shift that occurs during the entire lifespan of a page. Sites should strive to have a CLS score of 0.1 seconds or less.

Performance measurement and profiling

- https://github.com/alexanderteplov/computer-science/wiki/Measurement-and-profiling
- Core Web Vitals workflows with Google tools

Network optimizations (images, gzipping, bundling, HTTP2, etc)

- Website performance: How I've improved the performance of this website? - Meziantou's blog
- A Beginner's Guide to Website Speed Optimization
- 12 Techniques of Website Speed Optimization: Performance Testing and Improvement Practices

Improving user perception with layout placeholders, async/defer for not blocking the browser, long data lists handling

- How Browsers Work: Behind the scenes of modern web browsers - HTML5 Rocks
- Lazy loading - Web Performance | MDN
- Preload critical assets to improve loading speed

CPU bound operations optimizations

- Front-End Performance Checklist 2021 (PDF, Apple Pages, MS Word) — Smashing Magazine
- JavaScript Start-up Optimization | Web Fundamentals | Google Developers

Web workers, service workers, and worklets are all scripts that run on a separate thread to the browser's main thread. Where they differ is in where they are used and what features they have to enable these use cases. Worklets are hooks into the browser's rendering pipeline, enabling us to have low-level access to the browser's rendering processes such as styling and layout. Service workers are a proxy between the browser and the network. By intercepting requests made by the document, service workers can redirect requests to a cache, enabling offline access. Web workers are general-purpose scripts that enable us to offload processor-intensive work from the main thread.

- Web workers vs Service workers vs Worklets
- Using Web Workers - Web APIs | MDN
- Service Worker API

Memory leaks detection

JS is one of the so-called garbage collected languages. Main cause for leaks in garbage-collected languages is unwanted references. Common JS Leaks: accidental global variables, forgotten timers or callbacks, out of DOM references, closures, pieces of strings pointed to huge strings, which are considered removed. Most garbage collectors use an algorithm known as mark-and-sweep.

1. The garbage collector builds a list of "roots". Roots usually are global variables to which a reference is kept in code. In JavaScript, the "window" object is an example of a global variable that can act as a root.
2. All roots are inspected and marked as active (i.e. not garbage). All children are inspected recursively as well. Everything that can be reached from a root is not considered garbage.
3. All pieces of memory not marked as active can now be considered garbage. The collector can now free that memory and return it to the OS.

Modern garbage collectors improve on this algorithm in different ways, but the essence is the same: reachable pieces of memory are marked as such and the rest is considered garbage. Unwanted references are variables kept somewhere in the code that will not be used anymore and point to a piece of memory that could otherwise be freed.

- Memory Management - JavaScript | MDN
- 4 Types of Memory Leaks in JavaScript and How to Get Rid Of Them
- Inspect JavaScript ArrayBuffer with the Memory inspector - Chrome Developers
- Fixing memory leaks in web applications | Read the Tea Leaves

V8 hidden classes and inline caching techniques. To more efficiently work with classes dynamic properties V8 creates intermediate classes: every time you add a new property in runtime V8 cashes a hidden class and the reference to it (if there isn't already such a cached sample). So, if you add the same properties in a different order, there will be different classes in a cache chain (this affects performance). Characteristics of hidden classes:

1. Every object has a hidden class of its own.
2. A hidden class contains a memory offset for each property.
3. When a property is created dynamically, or a property is deleted or changed, another hidden class is created. The new hidden class keeps the information on the existing properties and at the same time, keeps the memory offset of the new property.
4. A hidden class knows which hidden class to refer to when a property is changed.
5. If an object gets a new property, the transition information of the object's hidden class is checked. If the transition information contains the condition identical to the property change, then the object changes its hidden class to the one defined in the transition information.

V8 takes advantage of **inline caching** (it relies upon the observation that repeated calls to the same method tend to occur on the same type of object). V8 maintains a cache of the type of objects that were passed as a parameter in recent method calls, and uses that information to make an assumption about the type of object that will be passed as a parameter in the future. If V8 is able to make a good assumption about the type of object that will be passed to a method, it can bypass the process of figuring out how to access the objects properties, and instead use the stored information from previous lookups to the objects hidden class. Whenever a method is called on a specific object, the V8 engine has to perform a look up to that object's hidden class to determine the offset for accessing a specific property. After two successful calls of the same method to the same hidden class, V8 omits the hidden class lookup and simply adds the offset of the property to the object pointer itself. For all future calls of that method, the V8 engine assumes that the hidden class hasn't changed, and jumps directly into the memory address for a specific property using the offsets stored from previous lookups; this greatly increases execution speed.

- How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code | by Alexander Zlatkov | SessionStack Blog
- V8 Hidden class - LINE ENGINEERING
- Javascript Hidden Classes and Inline Caching in V8

Event loop

- Event Loop
- Tasks, microtasks, queues and schedules - JakeArchibald.com
- How JavaScript works: Event loop and the rise of Async programming + 5 ways to better coding with async/await | by Alexander Zlatkov | SessionStack Blog

05. Web Application Design And Framework

SPA vs MPA pros/cons

- https://kb.epam.com/display/EPMCJSCC/MPA+vs+SPA
- Single Page Application (SPA) vs Multi Page Application (MPA) – Two Development Approaches | ASPER BROTHERS

SSR vs CSR pros/cons

- https://kb.epam.com/display/EPMCJSCC/SSR+vs+CSR
- SSR vs CSR - DEV Community

Micro-frontends

- https://github.com/alexanderteplov/computer-science/wiki/Micro-frontends
- Micro Frontends

Monorepos: A Guide to Monorepos for Front-end Code | Toptal

PWA (Progressive Web Application)

- Progressive Web Apps (PWA): Features, Architecture, Pros and Cons | AltexSoft
- PWA-приложения. Что это такое и для чего бизнесу создавать приложение из сайта — Маркетинг на vc.ru

CSS methodologies (BEM, OOCSS, SMACSS, Atomic CSS, CSS-in-JS, CSS Modules)

- 5 Methodologies for Architecting CSS
- CSS Methodologies
- A web designer's guide to CSS methodologies | Creative Blog

Angular vs React vs Vue vs etc. – how to choose the next project framework?

- https://kb.epam.com/display/EPMCJSCC/FE+Frameworks
- Choosing the Best Front-end Framework | Toptal
- take into consideration: community, GitHub stars, repos, forks, components; documentation; support, new versions; popularity trends (npm, google, StackOverflow); learning curve; features out of the box; extending with new features; scalable for big teams; versions backward compatibility; native/mobile apps support; bundle size; flexible/opinionated in project structure/configuration; ease of integration to existing apps (micro-frontends friendliness); production case studies; team background/level of knowledge; SSR support.

06. Framework Deep Dive

JS Core

The Modern JavaScript Tutorial
ECMAScript 6
JavaScript ES2016 Features With Examples | by Carlos Caballero | Better Programming
JavaScript ES2017 Features With Examples | by Carlos Caballero | Better Programming
JavaScript ES2018 Features With Examples | by Carlos Caballero | Better Programming
JavaScript ES2019 Features With Examples | by Carols Caballero | Better Programming

JavaScript ES2020 Features With Simple Examples | by Carlos Caballero | Better Programming
JavaScript ES2021 Features With Simple Examples | by Carlos Caballero | Better Programming

React

Pros and Cons of ReactJS - javatpoint
React lifecycle methods diagram
React Top-Level API
Introducing Hooks – React
React Patterns
React App Performance, Measuring React app performance
React Fiber
Introducing Concurrent Mode (Experimental) – React
Reconciliation, Reconciliation – React
Working with Redux: Pros and Cons
State Management
Testing Overview – React
Building reusable UI components with React Hooks
Server-Side Rendering in React using Next.js – How it Works & Implementation Example
Theming and Theme Switching with React and styled-components
How to scope your CSS/SCSS in React JS
How to translate your React app with react-i18next

Angular

Скринкаст по Angular
Artur Androsovych – Medium (whole blog)
Angular inDepth - Community of passionate Angular engineers (whole blog)
NgRx, RxJS (documentations)
Understanding hot vs cold Observables
Angular - NgModules
Dependency Injection in Angular - YouTube
Dependency injection in Angular
Angular - Dependency providers
Angular - Dependency injection in action
Angular - Hierarchical injectors
Angular Change Detection - How Does It Really Work?
RxJS pros and cons - your intro to JS's reactive programming library
The magic of RxJS sharing operators and their differences
Angular: понятное введение в NGRX
Angular vs. React: Which is Better and Why?
Upgrading from AngularJS to Angular
Benefits/drawbacks of Angular 8 in comparison with 1 (how to convince client to migrate)
Angular Performance Optimization Techniques
Increase performance of your Angular Applications
Understand and prevent the most common memory leaks in Angular application
Angular - DevTools Overview
Angular - Lifecycle hooks
Just-in-Time (JIT) and Ahead-of-Time (AOT) Compilation in Angular
Just in Time Compilation Explained
Angular - Ahead-of-time (AOT) compilation
Angular - Content projection, Проекция контента в Angular

07. Architecture

What is Software Architecture?

Software elements (modules); relations among them (connections and communication protocols); properties of both elements and relations (quality attributes); constraints.

Software architecture is the shared understanding that expert developers have of the system design; it's about the important stuff – whatever that is. This understanding includes all modules (elements), how they interact with each other, the environment in which they operate, and the principles used to design the software. In many cases, it can also include the software's evolution. Software architecture is designed with a specific mission which has to be accomplished without hindering the missions of other tools or devices.

Most of the proposed views generally belong to one of these types: module, component and connector (C&C), allocation. In a module view, the system is viewed as a collection of code units, each implementing some part of the system functionality. That is, the main elements in this view are modules. These views are code-based and do not explicitly represent any runtime structure of the system. Examples of modules are packages, a class, a procedure, a method, a collection of functions, and a collection of classes. The relationships between these modules are also code-based and depend on how code of a module interacts with another module.

In a C&C view, the system is viewed as a collection of runtime entities. That is, a component is a unit, which has an identity in the executing system. Objects (not classes), a collection of objects, and a process are examples of components. While executing, components need to interact with others to support the system services. Connectors provide means for this interaction (pipes, sockets, tyre, shared data, middleware, communication protocols).

An allocation view focuses on how the different software units are allocated to resources like the hardware, file systems, and people. That is, an allocation view specifies the relationship between software elements and elements of the environments in which the software system is executed. They expose structural properties like which processes run on which processor, and how the system files are organized on a file system.

Quality attributes: performance, interoperability, usability, reliability, availability, security, maintainability, modifiability, testability, scalability, reusability, supportability.

Constraints: topology, DBMS, hardware (bandwidth, servers with configurations etc.), operating system, used architectural style or pattern, development tools, team structure, schedule, legacy code.

Architecture frameworks: Zachman (what, how, where, when, why), DoDAF (Department of Defense Architecture Framework), TOGAF (The Open Group Architecture Framework). The goal of enterprise architecture is to translate business vision and strategy into effective enterprise. Enterprise architecture addresses among others the alignment between business, application, information and technology layers, usually in a top-down approach.

Architectural styles and patterns: client-server, multilayered architecture (presentation layer, application layer, business layer, data access layer), peer-to-peer, service-oriented architecture, cloud computing services.

Steps to define Architecture?

1) Identify Stakeholders (to communicate with): corporate functions, end user organizations, project organizations, system operations or external; then classify their positions and management approach. **Stakeholder** is an individual, group, or organization who may affect, be affected or perceive itself to be affected by a decision, activity, or outcome of a project.

2) Understand the problem, Functional Requirements: without a clear understanding of the problem, it's impossible to create an effective solution. In fact, many software products are considered unsuccessful because they didn't actually solve a valid business problem or have a recognizable return

on investment (ROI). Functional Requirements define **what a product must do**, what its features and functions are. Examples:

Authentication of a user when he/she tries to log into the system.

System shutdown in the case of a cyber-attack.

Verification email is sent to the user whenever he/she registers for the first time on some software system.

Typical functional requirements include: Business Rules, Transaction corrections, adjustments and cancellations, Administrative functions, Authentication, Authorization levels, Audit Tracking, External Interfaces, Certification Requirements, Reporting Requirements, Historical Data, Legal or Regulatory Requirements.

3) Identify design elements and their relationships (boundaries and context of the system): this means build a baseline for defining the boundaries and context of the system. Decomposition of the system into its main components is based on the functional requirements. The decomposition can be modeled by using a design structure matrix (DSM), which shows the dependencies between design elements without specifying the granularity of the elements. In this step, the first validation of the architecture is done by describing a number of system instances and this step is referred to as functionality based architectural design.

4) Evaluate the Architecture Design (Non-Functional Requirements, quality attributes, constraints): Each quality attribute is given an estimate, so in order to gather qualitative measures or quantitative data, the design is evaluated. It involves evaluating the architecture for conformance to architectural quality attributes requirements. If all the estimated quality attributes are as per the required standard, the architectural design process is finished. If not, then the third phase of software architecture design is entered: architecture transformation. However, if the observed quality attribute does not meet its requirements, then a new design must be created.

Non-functional requirements, not related to the system functionality, **rather define how the system should perform**. Examples:

Emails should be sent with a latency of no greater than 12 hours.

Each request should be processed within 10 seconds.

The site should load in 3 seconds when the number of simultaneous users are more than 10000.

Typical non-functional requirements are:

a) Performance and scalability. How fast does the system return results? How much will this performance change with higher workloads?

b) Portability and compatibility. Which hardware, operating systems, browsers, and their versions does the software run on? Does it conflict with other applications and processes within these environments?

c) Reliability, availability, maintainability. How often does the system experience critical failures? How much time is the system available to users against downtimes?

d) Security. How are the system and its data protected against attacks?

e) Localization. Does the system match local specifics?

f) Usability. How easy is it for a customer to use the system?

Constraints: often represented as a subset of quality attributes, are more concrete than common quality attributes. For instance: 1) the project will be developed in either ASP.NET or PHP; 2) the database will be SQL Server; 3) open-source products only shall be used for development and run time; 4) the system will be scalable; 5) the system will implement fault tolerant policies; 6) the system will be designed in order to minimize processing and response time. 4 to 6 are also quality attributes, but 1 to 3 aren't – they just state some constraint to take into account for analysis/design, development and/or runtime.

5) Prioritize quality attributes (trade-off analysis, suffer ones for others): it's possible to use a consistent approach for prioritization which is based on pairwise comparison of quality attributes with an analytic hierarchy process.

Trade-off analysis consists of gathering stakeholders together to analyze business drivers (system functionality, goals, constraints, desired non-functional properties) and extract quality attributes that are used to create scenarios. These scenarios are used in conjunction with architectural approaches and decisions to create an analysis of trade-offs, sensitivity points, and risks (or non-risks). This analysis can be converted to risk themes and their impacts whereupon the process can be repeated. To reach the optimum balance of product characteristics, you must identify, specify, and prioritize the pertinent quality attributes during requirements elicitation.

6) Transform the architecture design (quality attributes oriented – optimization of solutions using various patterns): the architectural design must be changed until it completely satisfies the quality attribute requirements. It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality. Further, a design is transformed by applying design operators, styles, or patterns. For transformation, take the existing design and apply design operators (decomposition, replication, compression, abstraction, resource sharing). Moreover, the design is again evaluated and the same process is repeated multiple times if necessary and even performed recursively. The transformations (i.e. quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively.

Patterns Transform Architecture:

- Candidate security patterns (account lockout, audit interceptor, DoS safety, safe data structure, secure logger)
- Patterns with high-level solutions (container managed security, dynamic service management, multilevel security)
- Patterns with hard-to-implement solutions (delaying routing, limited access, secure server proxy single threaded facade)
- Patterns with common solutions (partitioning, cryptographic transformations, hidden implementation, intercepting validator)

7) Document the results (to facilitate communication between stakeholders): architecture must be documented in a good amount of detail and should be presented in an accessible form for many different stakeholders. The architectural documentation is referring to the high-level description of the system, showing its fundamental principles of work. The primary purpose of this documentation is to correlate functional and non-functional requirements. Reasons to document the results:

- Whiteboard designs are not persistent.
- Explaining the principles of architecture to a wider audience gets difficult.
- The various decisions that drive the design could be forgotten and documenting them could help us get some rationale.
- Solid way to address stakeholders' concerns ahead of time.
- Easy to visualize and plan for many different needs.

Good Architecture principles

1) SOLID:

- **S** – Single Responsibility Principle: there should be only a single reason for any changes in a module; each module should be responsible to one, and only one, actor.
- **O** – Open-Closed Principle: software entities (classes, modules, functions, and so on) should be open to extension and closed to modification.
- **L** – Liskov Substitution Principle: all instances of a superclass can be substituted by instances of their subclasses. It means that superclass's preconditions can't be

strengthened, postconditions can't be weakened in subclasses, invariants should be preserved with subclasses. There should be a prohibition on changing immutable (at superclass) properties in subclasses.

- **I** – Interface Segregation Principle: no code parts should be forced to depend upon interfaces that they don't use; it's much better to use small and client-oriented interfaces (this provides simplification of code and creates a barrier preventing coupling to spare dependencies).
- **D** – Dependency Inversion Principle: high-level modules shouldn't import anything from low-level modules. Both should depend on abstractions (e.g., interfaces). Abstractions should not depend on details. Concrete implementations should depend on abstractions.

2) Separation of concerns (SoC): separate your application into different sections, and each section will address a separate concern (a set of information that affects the program's code). A concern can be as general as "the details of the hardware for an application", or as specific as "the name of which class to instantiate".

SoC for programming functions: avoid writing long complex functions, parts of the original algorithm should be exported and encapsulated in separate smaller functions with a private access level because they tend to be reusable.

SoC for modules: group functions under self-contained modules, each responsible for the fulfillment of a single set of tasks that have a clear logical correlation.

Benefits of loose coupling and high cohesion: better code clarity, better code reusability, better testability, faster project evolution, simple organization of simultaneous development.

SoC for the system's design: for a bunch of modules with distinct responsibilities and clear purpose there is a need to outline a global strategy to how the modules should refer to each other (otherwise there will be a system with entangled relations and hard-to-track data flows). Every existing architecture pattern provides this common strategy. The resulting set of modules within one layer has high cohesion based on the similar duties and the same level of abstraction, while communication and environment awareness between the layers is very much restricted to achieve loose coupling.

SoC for a website: organization of content (HTML), style and presentation (CSS), business logic – interaction and behavior (JavaScript).

SoC for a web server: **input layer** – responsible for accepting input HTTP requests, validating them for proper authentication and format, and then dispatching them to the correct logic function; **logic layer** – contains the algorithms which operate on the data in response to user input; **data access layer** – responsible for reading and writing in-memory representations of records to and from the database, as well as performing complex queries over the data.

3) System Service Components should be business focused. Security, communications, or system services (logging, profiling, configuration) should be abstracted in separate components.

4) Single Responsibility principle: every module for a single functionality. Dependency Injection (for primary activities within app) and Event Driven Architecture (for secondary activities within app) principles derive from SRP (their usage facilitates the reduction of code's redundancy).

5) Principle of Least Knowledge (Law of Demeter): an object should have limited knowledge about other objects. The code inside some method should express knowledge only of its surroundings (the object that owns this method, method's arguments, objects that are held in instance variables, locally created objects, accessible global objects). It's very close to One Dot Principle (code shouldn't access data in some object throughout more than one dot, e.g. *user.getAccount().getBalance(). substractSum(invoiceTotal)*). No business logic not belonging to some module should be placed into this module.

6) DRY ("don't repeat yourself"): every piece of knowledge must have a single, unambiguous, authoritative representation within a system. It means that it's not only about copy and paste code –

yes, this is also included – but goes beyond that. It's also about having different code that does the same thing. Maybe you can have different code in two or more places, but they do the same thing in different ways, this also should be avoided.

7) KISS ("keep it simple, stupid"): this principle says about to make your code simple (without unnecessary complexity) because it's easier to maintain and understand. Also, it requires separate responsibilities of classes and layers within the app.

8) Minimize upfront design: only design what is necessary. YAGNI ("You ain't gonna need it"), think ROI. It's similar to KISS principle, once that both of them aim for a simpler solution. The difference between them it's that YAGNI focuses on removing unnecessary functionality and logic, and KISS focuses on complexity. Ron Jeffries, one of the co-founders of XP: "Always implement things when you actually need them, never when you just foresee that you need them".

9) Composition Over Inheritance while reusing the functionality: this is a principle that classes should achieve polymorphism and code reuse by composition ("HAS-A"), instead of through inheritance ("IS-A"). In most cases "HAS-A" is more semantically correct than "IS-A" between classes because of the composition's flexibility (you can change implementation of class at run-time by changing included object, but you can't change behavior of base class at run-time). Also, there is no conflict between methods/properties names, which might occur with inheritance. Though the behavior of the system may be harder to understand just by looking at the source code, since it's more dynamic and more interaction between classes happens in run-time, rather than compile time.

10) Identify components and group them in logical layers: the actual idea of separating a project into layers suggests that this separation of concerns should be achieved by source code organization. This means that apart from some guidance to what concerns we should separate, the Layered Architecture tells us nothing else about the design and implementation of the project. This implies that we should complement it with some other architectural processes, such as some upfront design, daily design sessions, or even full-blown Domain-Driven Design.

11) Define the communication protocol, data format between layers: communication protocols belong to some layers at OSI model, data formats are defined with used protocols and opportunities of implemented layers (more details at "Communication protocols").

12) Design exceptions and exception handling mechanism: exception handling is the process of responding to the occurrence of exceptions during the execution of a program. Exception handling is facilitated by specialized programming language constructs, hardware mechanisms like interrupts, or OS inter-process communication facilities like signals. Alternative approach to exception handling is error checking, which maintains normal program flow with later explicit checks for contingencies reported using special return values, an auxiliary global variable, or floating-point status flags.

13) Naming Convention: set of rules for choosing the character sequence to be used for identifiers which denote other entities in source code (variables, types, functions) and documentation. Reasons for naming convention's usage: reducing the effort needed to read and understand source code, focusing on more important issues (than syntax and naming standards) during code reviews, simplification of quality review tools' work and functionality.

08. Patterns

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that can be customized to solve a recurring design problem in a program. Most patterns are described very formally so they can be reproduced in different contexts. Sections that are usually present in a pattern description: intent (brief description), motivation, structure of classes, code examples.

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

- **Factory method** provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. Pattern's applicability: exact types aren't known beforehand; provision of program's flexible extension; saving system resources by reusing existing objects.
- **Abstract Factory** allows the production of related objects' families without specifying their concrete classes. Pattern's applicability: there is a need to work with various families of related products that aren't known beforehand or might be created afterwards; there is a set of factory methods that blur primary responsibility of class.
- **Builder** allows a complex construction of objects step by step (in other words, to produce different representations of some object using the same construction code). Pattern's applicability: there are no overloading methods in programming language, construction of complex objects such as Composite trees.
- **Prototype** allows copying of existing objects without making code dependent on their classes. Pattern's applicability: reduction of subclasses that only differ in the way they initialize their respective objects.
- **Singleton** restricts instantiation of a class to a single object (if an instance is already existing, class simply returns a reference to that object). Pattern's applicability: provision of shared functionality, stricter control over global variables.

Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

- **Adapter** allows classes/objects with incompatible interfaces to collaborate. Pattern's applicability: reuse several existing subclasses that lack some common functionality that can't be added to the superclass.
- **Bridge** provides a split of a large class or a set of closely related classes into abstraction and implementation hierarchy which can be developed independently of each other. Pattern's applicability: several variants of usage, extension in independent dimensions, switching of implementations at runtime.
- **Composite** provides a composition of objects into tree structures and following work with these structures as if they were individual objects. Pattern's applicability: implementation of tree-like objects, treating of simple and complex elements uniformly.
- **Decorator** allows to attach new behaviors to objects by placing these objects inside special wrapper objects that contain behaviors. Pattern's applicability: dynamic addition of behavior to existing classes.
- **Facade** provides a simplified interface to a library, a framework, or any other complex set of classes. Pattern's applicability: limited but straightforward interface to a complex subsystem, layered structure of subsystem.
- **Flyweight** allows fitting more objects into available amounts of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object. Pattern's applicability: support of objects which barely fit into available RAM.
- **Proxy** provides a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object. Pattern's applicability: lazy initialization, access control, local execution of remote service, logging requests, caching responses, smart reference.

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

- **Chain of responsibility** passes requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. Pattern's applicability: different kinds of requests are expected, but exact types

and their sequences are unknown beforehand, execution of several handlers in a particular order, set of handlers and their order are supposed to change at runtime.

- **Command** turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as method arguments, delay or queue the request's execution, and support undoable operations. Pattern's applicability: parametrization of objects with operations, implementation of reversible operations, desire to queue operations, schedule their execution, or execute them remotely.
- **Iterator** accesses the collection's elements sequentially without exposing its underlying representation. Pattern's applicability: desire to reduce duplication of the traversal code across the app, types of data structures are unknown beforehand.
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods. Pattern's applicability: simplification of classes' changes, reuse of some basic behavior in various contexts.
- **Memento** provides the ability to restore an object to its previous state (undo). Pattern's applicability: necessity to produce snapshots of the object's state, direct access to the object's fields/getters/setters violates its encapsulation.
- **Observer** is a publish/subscribe pattern, which allows a number of observer objects to see an event. Pattern's applicability: changes to some object's state may require changing other objects (actual set is unknown beforehand), observation of objects for a limited time or in specific classes.
- **State** allows an object to alter its behavior when its internal state changes. Pattern's applicability: object behaves differently depending on its current state (number of states is enormous or there are massive conditionals which determine behavior of class), reduction of duplicate code.
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime. Pattern's applicability: usage of different algorithms at runtime, isolation of business logic from implementation.
- **Template method** defines the algorithm's skeleton as an abstract class, allowing its subclasses to provide concrete behavior. Pattern's applicability: extrinsic extension of algorithm's particular steps.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object. Pattern's applicability: operation's execution at some complex set of objects, clean up the business logic of auxiliary behaviors.

MVC is an architectural design pattern that encourages improved application organization through a SoC. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic and user-input. Models are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require. When a model changes, it will typically notify its observers (views) that a change has occurred so that they may react accordingly.

Views typically observe models and are notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as "dumb" given that their knowledge of models and controllers in an application is limited. Users are able to interact with views and this includes the ability to read and edit (i.e get or set the attribute values in) models. As the view is the presentation layer, we generally present the ability to edit and update in a user-friendly fashion. A view is an object which observes a model and keeps the visual representation up-to-date. A template *might* be a declarative way to specify part or even all of a view object so that it can be generated from the template specification. It is also worth noting that in classical web development, navigating between independent views required the use of a page refresh. In SPA once data is fetched from a server, it can

simply be dynamically rendered in a new view within the same page without any such refresh being necessary.

The actual task of updating the model falls to controllers. Controllers are an intermediary between models and views which are classically responsible for updating the model when the user manipulates the view. Remember that the controllers fulfill one role in MVC: the facilitation of the Strategy pattern for the view. In the Strategy pattern regard, the view delegates to the controller at the view's discretion. So, that's how the strategy pattern works. The view could delegate handling user events to the controller when the view sees fit. The view *could* delegate handling model change events to the controller if the view sees fit, but this is not the traditional role of the controller.

This separation of concerns in MVC facilitates simpler modularization of an application's functionality and enables:

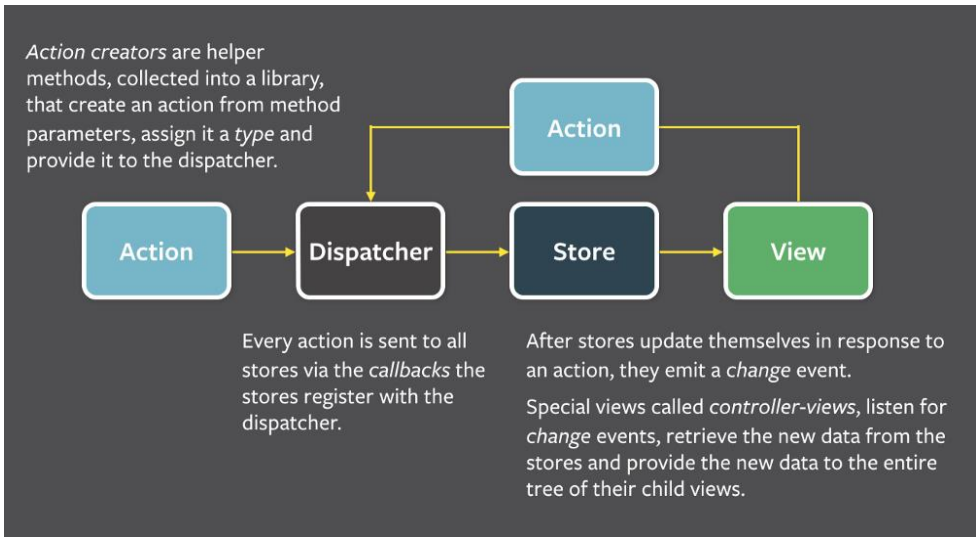
- Easier overall maintenance. When updates need to be made to the application it is very clear whether the changes are data-centric, meaning changes to models and possibly controllers, or merely visual, meaning changes to views.
- Decoupling models and views mean that it is significantly more straight-forward to write unit tests for business logic.
- Duplication of low-level model and controller code (i.e what we may have been using instead) is eliminated across the application
- Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user-interfaces to work simultaneously.

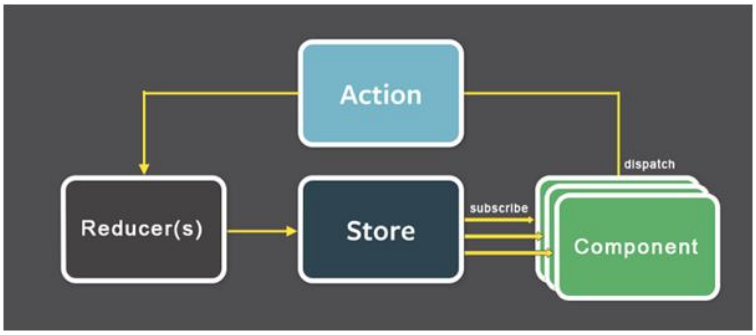
MVP is a derivative of the MVC design pattern which focuses on improving presentation logic. Solicited by a view, presenters perform any work to do with user requests and pass data back to them. In this respect, they retrieve data, manipulate it and determine how the data should be displayed in the view. In some implementations, the presenter also interacts with a service layer to persist data (models). Models may trigger events but it's the presenter’s role to subscribe to them so that it can update the view. In this passive architecture, we have no concept of direct data binding. Views expose setters which presenters can use to set data. The benefit of this change from MVC is that it increases the testability of our application and provides a cleaner separation between the view and the model. This isn't without its costs as the lack of data binding support in the pattern can often mean having to take care of this task separately.

MVVM attempts to more clearly separate the development of UI from that of the business logic and behavior in an application. A lot of implementations of this pattern make use of declarative data bindings to allow a separation of work on Views from other layers. This facilitates UI and development work occurring almost simultaneously within the same codebase. UI developers write bindings to the ViewModel within their document markup, where the Model and ViewModel are maintained by developers working on the logic for the application. The ViewModel might be looked upon as more of a Model than a View but it does handle most of the View's display logic. It may also expose methods for helping to maintain the View's state, update the model based on the actions on a View and trigger events on the View. But data-bindings in non-trivial applications can create a lot of book-keeping.

In MVC, the View sits on top of our architecture with the controller beside it. Models sit below the controller and so our Views know about our controllers and controllers know about Models. Here, our Views have direct access to Models. Exposing the complete Model to the View however may have security and performance costs, depending on the complexity of our application. MVVM attempts to avoid these issues. In MVP, presenters sit at the same level as views, listening to events from both the View and model and mediating the actions between them. Unlike MVVM, there isn’t a mechanism for binding Views to ViewModels, so we instead rely on each View implementing an interface allowing the Presenter to interact with the View. MVVM consequently allows us to create View-specific subsets of a

Model which can contain state and logic information, avoiding the need to expose the entire Model to a View. Unlike MVP's Presenter, a ViewModel is not required to reference a View. The View can bind to properties on the ViewModel which in turn expose data contained in Models to the View. As we've mentioned, the abstraction of the View means there is less logic required in the code behind it. One of the downsides to this however is that a level of interpretation is needed between the ViewModel and the View and this can have performance costs. The complexity of this interpretation can also vary – it can be as simple as copying data or as complex as manipulating them to a form we would like the View to see. MVC doesn't have this problem as the whole Model is readily available and such manipulation can be avoided.



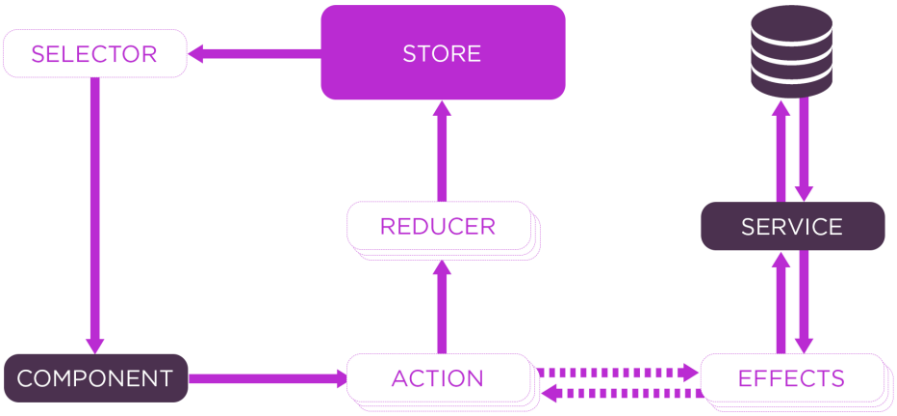


In the Flux architecture, when a user clicks on something, the view creates actions. Action can create new data and send it to the dispatcher. The dispatcher then dispatches the action result to the appropriate store. The store updates the state based on the result and sends an update to the view. In Redux architecture, an application's event is denoted as an Action, which is dispatched to the reducer, the pure function. Then reducer updates the centralized store with new data based on the kind of action it receives. Store creates a new state and sends an update to the view. At that time, the view was recreated to reflect the update. The significant benefit of a unidirectional approach (which is used in Flux/Redux) is that since the data flows through your application in a single direction you can have better control over it. In Flux, the logic of changes in the data based upon the actions is mentioned in its appropriate Store. The Store in the Flux app also possesses the flexibility to decide what parts of your data to expose publicly. In Redux the logic remains in the reducer function, which receives the previous state and one action, then returns the new state.

Key concepts of NgRx

- Actions describe unique events that are dispatched from components and services.
- State changes are handled by pure functions called reducers that take the current state and the latest action to compute a new state.

- Selectors are pure functions used to select, derive and compose pieces of state.
- State is accessed with the Store, an observable of state and an observer of actions.
- All actions that are dispatched within an application state are always first processed by the reducers before being handled by the effects of the application state.
- Local state has to be initialized, but it can be done lazily.
- Local state is typically tied to the life-cycle of a particular component and is cleaned up when that component is destroyed.
- Users of ComponentStore can update the state through setState or updater, either imperatively or by providing an Observable.
- Users of ComponentStore can read the state through select or a top-level state$. Selectors are very performant.
- Users of ComponentStore may start side-effects with effect, both sync and async, and feed the data both imperatively or reactively.



JS module systems (ES6+ modules, CommonJS)

Modules are an integral piece of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized. In JavaScript, there are several options for implementing modules: object literal notation, the module pattern, asynchronous module definition (AMD) modules, CommonJS modules, ECMAScript Harmony modules.

The AMD module format itself is a proposal for defining modules where both the module and dependencies can be asynchronously loaded. It has a number of distinct advantages including being both asynchronous and highly flexible by nature which removes the tight coupling one might commonly find between code and module identity. The first two concepts worth noting about AMD are the idea of a *define* method for facilitating module definition and a *require* method for handling dependency loading. The dependencies argument represents an array of dependencies which are required by the defining module and the third argument ("definition function" or "factory function") is a function that's executed to instantiate the module.

CommonJS is a project with the goal to establish conventions on the module ecosystem for JavaScript outside of the web browser. The primary reason for its creation was a major lack of commonly accepted forms of JavaScript module units which could be reusable in environments different from that provided by conventional web browsers running JavaScript scripts (web servers or native desktop applications). CommonJS's module specification is widely used today, in particular for server-side JavaScript programming with Node.js. CommonJS module is a reusable piece of JavaScript which exports specific objects made available to any dependent code. Unlike AMD, there are typically no function wrappers around such modules. CommonJS modules basically contain two primary parts: a free variable named *exports* which contains the objects a module wishes to make available to other modules and a *require* function that modules can use to import the exports of other modules (exports.foo = foo, var lib = require("package/lib")).

ES modules use *import* and *export* statements for similar (but not identical) functionality. *import* declarations bind modules' exports as local variables and may be renamed to avoid name collisions or conflicts. *export* declarations declare that a local-binding of a module is externally visible such that other modules may read the exports but can't modify them (modules may export child modules but not defined elsewhere; their external name can differ from their local names, operator *as*). Dynamic imports (ES10): *import*(*module*) – returns Promise.

09. Code Quality

Tools and techniques for ensuring code quality

Functional code quality means following or meeting functional requirements. It's about what the code does. Activities that ensure functional code quality include unit testing and functional testing.

Structural code quality means adhering to project-specific guidelines, maintaining clean code, and minimizing unnecessary details. It's about how the code looks. Activities that ensure structural code quality include static code analysis and code review.

Key Code Quality Aspects to Measure:

- Reliability measures the probability that a system will run without failure over a specific period of operation. It relates to the number of defects and availability of the software. Number of defects can be measured by running a static analysis tool. Software availability can be measured using the mean time between failures.
- Maintainability measures how easily software can be maintained. It relates to the size, consistency, structure, and complexity of the codebase. You can't use a single metric to ensure maintainability. Some metrics you may consider to improve maintainability are the number of stylistic warnings and Halstead complexity measures. Both automation and human reviewers are essential for developing maintainable codebases.
- Testability measures how well the software supports testing efforts. It relies on how well you can control, observe, isolate, and automate testing, among other factors. Testability can be measured based on how many test cases you need to find potential faults in the system. Size and complexity of the software can impact testability. So, applying methods at the code level facilitates testability's improvement.
- Portability measures how usable the same software is in different environments. It relates to the platform's independence. It's important to regularly test code on different platforms, rather than waiting until the end of development. It's also a good idea to set your compiler warning levels as high as possible – and use at least two compilers. Enforcing a coding standard also helps with portability.
- Reusability measures whether existing assets can be used again. Assets are more easily reused if they have characteristics such as modularity or loose coupling. Reusability can be measured by the number of interdependencies. Running a static analyzer can help you identify these interdependencies.

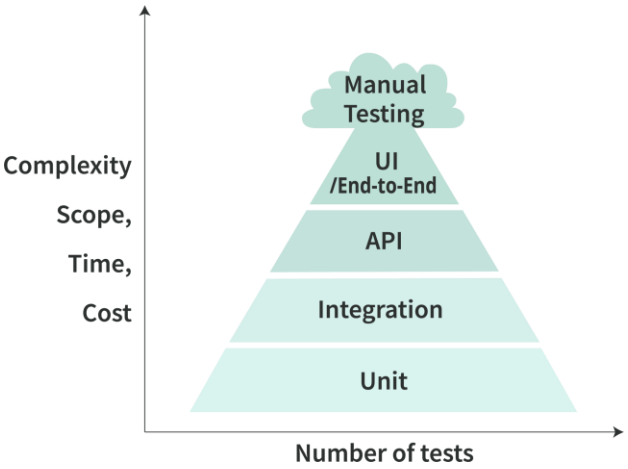
Code Quality Metrics to Use: defect metrics (identification of the stage in which the defect originates, number of open defect reports, time to identify and correct defects, defect density at code); complexity metrics (cyclomatic complexity, program vocabulary, program length, calculated program length, volume, difficulty, effort).

Quality ladder: correctness, efficiency, readability, extensibility.

Automated code analysis is a tool for achieving structural code quality and enforcing coding standards. It analyzes the program code against a predefined set of rules and best practices via a fully automated process, and subsequently provides objective feedback. SonarQube (formerly Sonar) is an open-source platform for continuous inspection of code quality that delivers one of the best static code analyzers on the market for Java, C#, C/C++, and etc. IntelliJ IDEA and Visual Studio can run static

code analysis checks immediately after a line of code is written, which helps developers fix violations quickly. Fortify is a static code analyzer that performs automated security testing, enabling teams to resolve issues faster.

Test pyramid (all types of tests)



For the test pipeline to be efficient, each stage must operate in a way that enables the next stage to operate correctly. The critical element of efficiency is a balanced test suite (set of automated test scripts). Each layer of the testing pyramid has specific attributes and is like a tool which can be used to conduct automated testing. Efficient automated testing should include multiple layers of tests (otherwise you won't be able to meet a client's expectations of high-quality product for a reasonable cost). Different test layers have different execution speeds, maintainability, complexity, and scopes. Lower-layer tests run first and most frequently. Higher-layer tests are slower, more complex, and harder to maintain; therefore, the cost of the higher-layer automated tests is much higher, and the scope is wider. The higher the test layers, the wider the scope and the more expensive automated testing becomes.

1. UI testing checks that entering a comment and clicking the "submit" button results in the comment text appearing on the wall.
2. API testing checks that if the user has access to add comments, the http response code is 201 Created, indicating a success status.
3. Integration testing checks that both the wall and audit services record the same comment, created date, and author.
4. Unit testing checks that a comment longer than 200 characters can't be saved.
5. Above all the layers of the Testing Pyramid is manual exploratory testing that helps discover new ways of how end users might use an application.

	Unit test	**Integration Test**	**System Test**	**E2E Test**
Scope	Modules, APIs	Modules, interfaces	Application, system	All subsystems, network dependencies, services and databases
Size	Tiny	Small to medium	Large	X-Large
Environment	Development	Integration test	QA test	Production like
Type of data	Mock data	Test data	Test data	Copy of real production data
System under test	Isolated unit test	Interfaces and flow data between the modules	Particular system as a whole	Application flow from start to end
Scenarios	Developer perspectives	Developers and IT pro tester perspectives	Developer and QA tester perspectives	End-user perspectives

When	After each build	After unit testing	After unit and integration testing	After system testing
Automated or manual	Automated	Both	Both	Manual

Code coverage is a simple metric often used to locate parts of the codebase that do not have unit tests. Code coverage is calculated as a ratio between the number of code lines executed during the unit test and the total number of lines in the code. Code coverage metrics may be useful to track the high-level project trend of unit testing activities (e.g., whole suite code coverage should not decrease). It may also help your team identify areas of the project that do not have unit tests. Code coverage doesn't give any insight into what code is tested or the quality of the executed tests. An extreme example is a test suite without a single assertion that may have code coverage of 99+%. Using this kind of test suite is a waste of time since it just executes the code and does not test it.

Unit tests (what is a good unit test? FIRST principles: understandable and readable, independent and isolated, repeatable and fast, should be well structured)

F (fast) – tests should be fast-running. A whole suite of unit tests should take seconds to run. The faster the tests, the more of them you can have in the suite, and the more often you can run them. When tests run slowly, your team will not run them frequently. As a result, you may not find problems early enough to fix them easily, which limits your ability to clean up the code, resulting in a gradual deterioration of code quality.

I (independent) – tests should not be dependent on each other. One test should not set the conditions for the next. Your team members should be able to run each test independently and in any order. When tests depend on each other, the first one to fail causes a cascade of downstream failures, making a diagnosis difficult and hiding downstream defects.

R (repeatable) – tests should be repeatable in any environment. If unit tests pass when running one-by-one but fail when running the whole test suite, or if they pass on your development machine but fail on the continuous integration server, there's a design flaw. Your team should be able to successfully run the tests on the production environment, QA environment, and laptops so there's never an excuse not to do it.

S (self-validating) – tests should have a boolean output and either pass or fail. The same test that fails now and passes later is flaky and compromises the whole testing suite. Flaky tests lead to negative consequences. Developers stop trusting tests and start ignoring them, and it becomes challenging to identify non-flaky tests that fail in a sea of flaky tests. You should not have to read through a log file or manually compare two text files to determine if a test passes. If they are not self-validating, then failure becomes subjective, and running tests requires a long manual evaluation.

T (timely) – tests should be written in a timely manner (before or at the same time as the production code). Testing post facto requires developers on your team to refactor the working code and make additional efforts to have tests fulfilling FIRST principles.

1. **Maintainable**: test code is considered just as important as production code (if you want your team's production code to be easily maintained, you must keep unit tests easy to maintain as well).
2. **Isolated from external dependencies** (e.g., database, file system, environment settings, network), otherwise it can fail for environmental reasons that have nothing to do with production code.
3. **Properly targeted**: a successful test suite targets only the essential parts of the production codebase on a project. For this reason, it is important to differentiate an application's domain model from the rest. Based on implemented logic, production code is classified into next types:

a. Practically unit-testable (core functions: business calculations, algorithms, reused code, etc.)
b. Unit-testable in theory (multi-threaded code, application bootstrapping, auto generated code, UI controls/styles, unstable prototype code, etc.)
c. Intentionally unit-untestable (interactions with external dependencies, boilerplate and trivial code: getter/setter, setup wiring, annotations, etc.)

TDD (Test Driven Development) means writing a test that fails because the specified functionality doesn't exist, then writing the simplest code that can make the test pass, then refactoring to remove duplication, etc. You repeat this Red-Green-Refactor loop over and over until you have a complete feature. BDD (Behavior Driven Development) means creating an executable specification that fails because the feature doesn't exist, then writing the simplest code that can make the spec pass. You repeat this until a release candidate is ready to ship. TDD cycle:

1. Add a test.
2. Run all tests. The new test should fail.
3. Write the simplest code that passes the new test. Inelegant or hard code is acceptable, as long as it passes the test.
4. All tests should now pass. If any fail, the new code must be revised until they pass.
5. Refactor as needed, using tests after each refactoring to ensure that functionality is preserved.

Differences between TDD and BDD

1. TDD is for unit testing. BDD is mostly for integration and e2e testing.
2. TDD implements in the codebase. BDD implements in pseudocode understandable for all the participants.
3. In TDD, tests are written by developers. In BDD several team roles could be evolved: QA engineers, business analysts, developers, etc.
4. TDD asserts modules functioning, BDD asserts user scenarios.

Code smells categories

1. Bloaters are code, methods and classes that have increased to such gargantuan proportions that they're hard to work with (long method, large class, primitive obsession, long parameter list, data clumps).
2. Object-oriented abusers (switch statements, temporary field, refused bequest, alternative classes with different Interfaces).
3. Change preventers mean making changes in other places beyond the initial one (divergent change, shotgun surgery, parallel inheritance hierarchies).
4. Dispensables (comments, duplicate code, lazy class, data class, dead code, speculative generality).
5. Couplers (feature envy, inappropriate intimacy, message chains, middle man).

Common tools and metrics for measure code quality

https://www.sonarqube.org/
https://www.atlassian.com/software/crucible
https://www.jetbrains.com/upsource/
https://www.phacility.com/phabricator/
https://www.reviewboard.org/
https://eslint.org/
https://jshint.com/
https://www.perforce.com/products/klocwork
https://codeclimate.com/
https://www.pylint.org/
https://www.codacy.com/

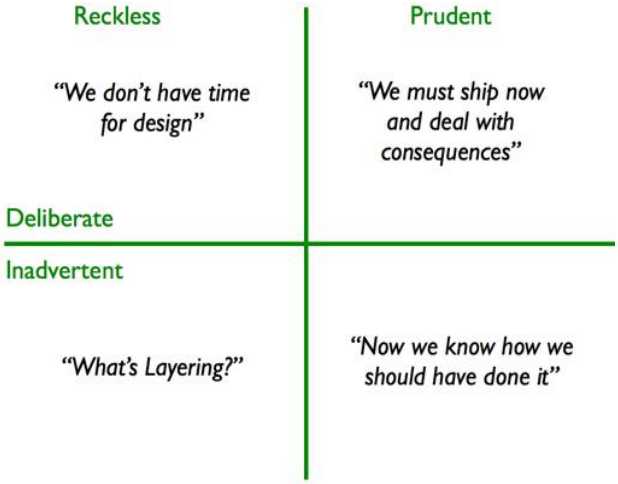
A code review checklist ensures that:

- Reviewers check all the necessary steps without skipping something important.
- The quality of each review is approximately equal, despite each reviewer's background.

- Newcomers to a project remember/follow all the steps of the process.
- It's easier to introduce a new practice/step that everyone should be aware of and follow. Highlighting it in a shared document makes it hard to forget.

Steps of code review checklist's creation: secure approval to create a code review checklist; describe in detail the process of development; provide improvement suggestions.
Code review types: peer review, specialist's review, instant code review.
Key areas of code review: functional correctness (business logic), structural design, readability / complexity, test correctness, non-functional hidden implications.

Process of handling technical debt, tools for managing technical debt



Technical debt reflects the implied cost of additional rework caused by choosing a limited solution now instead of using a better approach that would take longer. It's calculated as a ratio of the cost to fix a software system [Remediation Cost] to the cost of developing it [Development Cost].

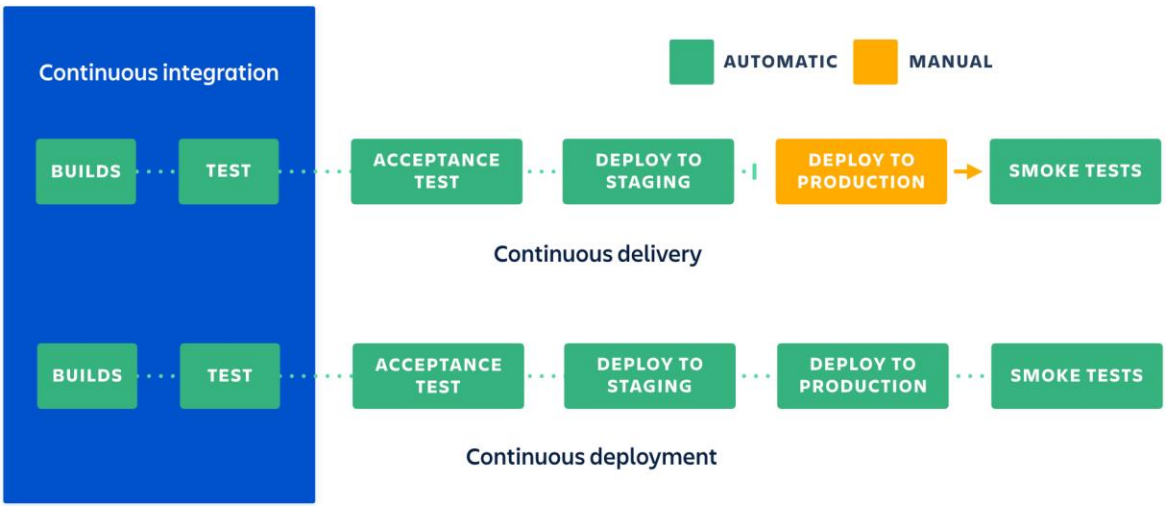
Technical Debt Ratio = (Remediation Cost / Development Cost) x 100%

Both remediation and development costs are represented in hours. Generally,no one wants a high Technical Debt Ratio [TDR], some teams favor values less than or equal to 5%. *Tools to manage technical debt*: documentation, project management tools (Jira), static analyser, linter, test coverage, technical debt management.

Handling of technical debt

1. Track: report tech debt impact from your workflow.
2. Prioritise: use insights to identify high value refactoring work.
3. Fix: address tech debt blocking your roadmap.
4. Measure: create tech debt KPIs and reports for your progress.

10. CI/CD/CD, GIT



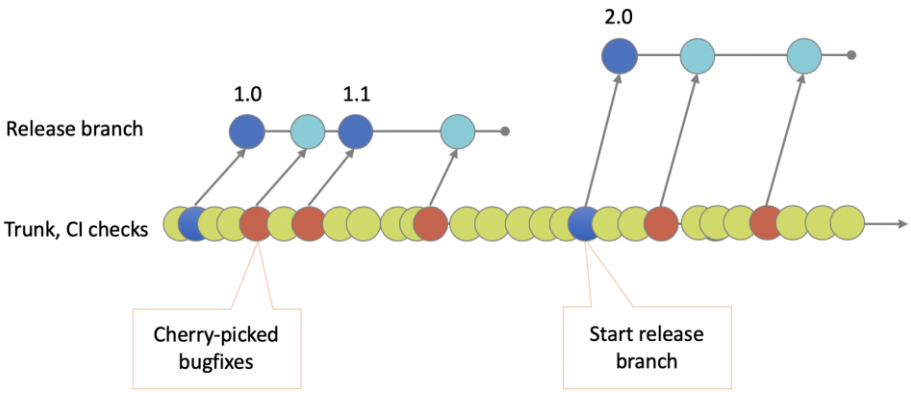
Continuous integration means integration of team's work frequently by collaborating on a shared codebase, merging disparate code changes into a version control system, and automatically creating and testing builds. It requires individual developers working on the same code to continually commit their code branches to a shared repository, where they are merged into the main build. Costs: writing

of automated tests, CI server, frequent merging of changes. Gains: less bugs get shipped to production, easy build of release, less switching among tasks, reduced testing costs.
Continuous delivery means that software can be released to production at any time by automatically deploying builds to non-production environments after the CI process completes. Costs: strong foundation in CI, sufficient test covering, automated deployments, embrace of feature flags. Gains: simplicity of deploying process, more frequent releases, less pressure on decisions for small changes.
Continuous deployment takes things one step further, meaning every change automatically gets put into production without any operator intervention. Costs: high quality of test suite, keeping up of documentation with the pace of deployments, feature flags as an inherent part of releasing significant changes. Gains: no need to pause development for releases (pipelines are triggered automatically for every change), releases are less risky and easier to fix because of small changes' batches, a continuous stream of improvements and quality's increasing.

Preparing pre-hooks, commit styles, CI/CD setup and configuration, linters rules

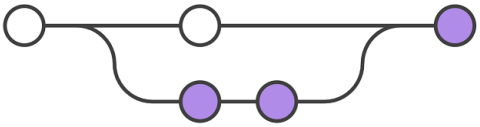
1. Create a project dir
2. Init git repository
3. Add .gitignore
4. Init npm package (create package.json)
5. Install via npm and configure Webpack (webpack.config.js)
6. Install via npm and configure ESLint (.eslintrc.js)
7. Install via npm and configure Stylelint (.stylelintrc.js)
8. Install via npm and configure TSLint (tslint.json)
9. Install via npm and configure Prettier (.prettierrc.json)
10. Install CI/CD automation tool and configure pipeline
11. Add hooks
12. Run unit tests over changes and prettier with pre-commit
13. Run linters with pre-push
14. Run all unit tests, integration tests, e2e test, build and deploy with post-push
15. Configure hooks to send notifications to developers and stakeholders

Branching strategy types

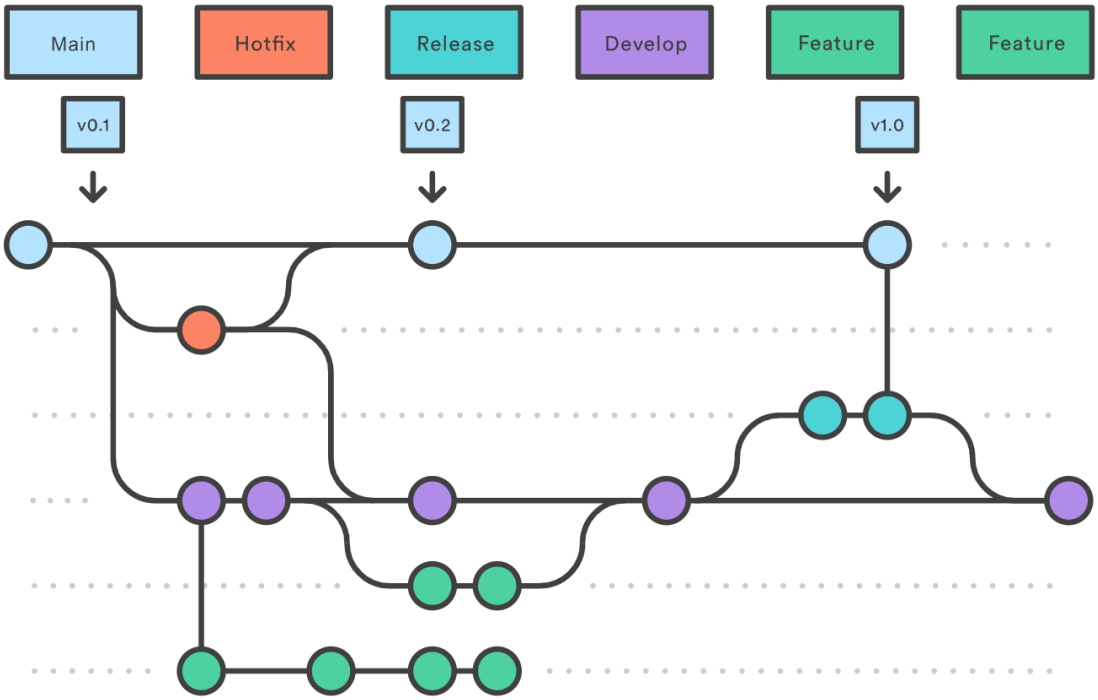


Trunk (mainline) based flow / Release Flow / One Flow: branch is created for a release, but only release engineers commit to the release branches. They may also cherry-pick individual commits from the mainline to the release branch if there is a desire to do so (in case of bugfix). This branching model requires much discipline, because a faulty commit almost instantly affects other developers. To maintain the mainline in a constantly healthy state, automatic pre-commit verifications and code reviews are frequently used. The state of the mainline codebase is generally monitored by a dedicated machine.
Feature Branch Workflow: all feature development should take place in a dedicated branch instead of the main branch. This makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. Encapsulating feature development also makes it possible to leverage

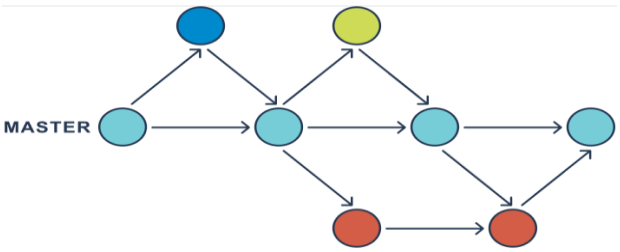
pull requests, which give the opportunity to sign off on a feature before it gets integrated into the official project. The Feature Branch Workflow can be leveraged by other high-level Git workflows.



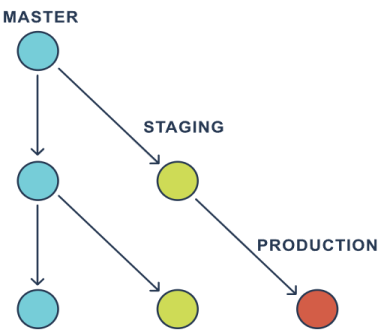
Git Flow has numerous, longer-lived branches and larger commits. Under this model, developers create a feature branch and delay merging it to the main trunk branch until the feature is complete. These long-lived feature branches require more collaboration to merge and have a higher risk of deviating from the trunk branch. Gitflow can be used for projects that have a scheduled release cycle and for the DevOps best practice of continuous delivery. It assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases.



GitHub flow: a simpler alternative of Git flow. It has only feature branches and a master branch. Anything in the master branch is deployable, so to work on something new, create a descriptively named branch. When it's ready, you create a PR. After someone else has reviewed and signed off on the feature, it deployed, got feedback, and merged into master.



GitLab flow suggests creating environment branches like staging and production. When someone wants to deploy to staging they create a merge request from the master branch to the pre-production branch. And going live with code happens by merging the staging branch into the production branch. This workflow, where commits only flow downstream, ensures that everything has been tested on all environments.



Forking Workflow: instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. The Forking Workflow is most often seen in public open source projects. The main advantage is that contributions can be integrated without the need for everybody to push to a single central repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

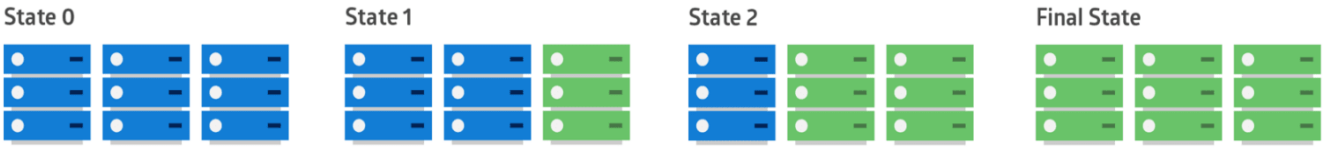
1. A developer 'forks' an 'official' server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A Git remote path for the 'official' repository is added to the local clone.
4. A new local feature branch is created.
5. The developer makes changes on the new branch.
6. New commits are created for the changes.
7. The branch gets pushed to the developer's own server-side copy.
8. The developer opens a PR from the new branch to the 'official' repository.
9. PR gets approved for merge and is merged into the original server-side repository.

Release strategies

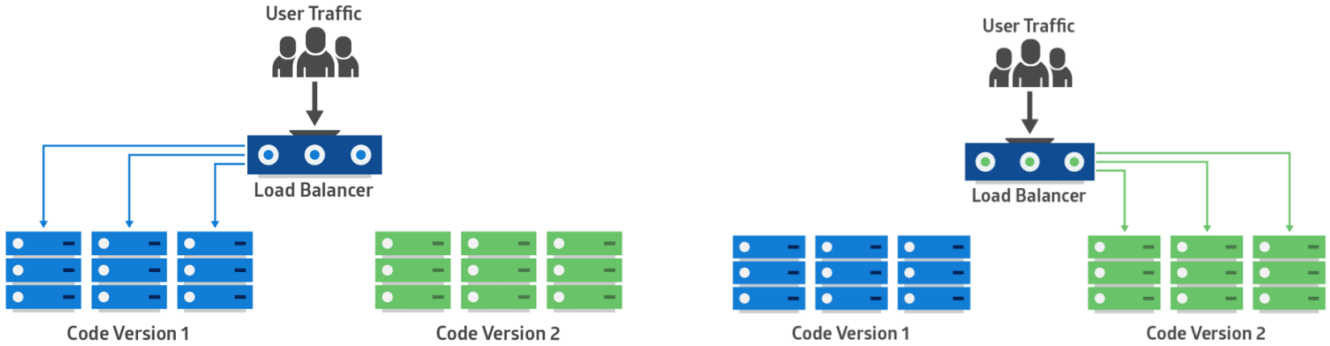
Big Bang deployment: update the app in one fell swoop. It can be suitable for non-production systems or vendor-packaged solutions like desktop applications. Main characteristics:

1. All major pieces packaged in one deployment.
2. Largely or completely replacing an existing software version with a new one.
3. Deployment usually results in long development and testing cycles.
4. Assuming a minimal chance of failure as rollbacks may be impossible or impractical.
5. Completion times are usually long and can take multiple teams' efforts.
6. Requiring action from clients to update the client-side installation.

Rolling (phased) deployment: an application's new version gradually replaces the old one. The actual deployment happens over a period of time. During that time, new and old versions will coexist without affecting functionality or user experience. This process makes it easier to roll back any new component incompatible with the old components.



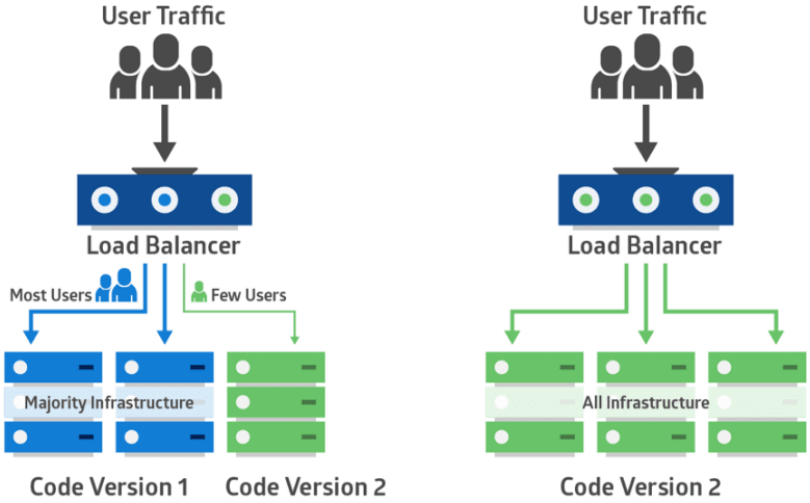
Blue-Green, Red-Black or A/B Deployment: one is the currently-running production environment receiving all user traffic (depicted as Blue). The other is a clone of it, but idle (Green). Both use the same database back-end and app configuration. The new version of the application is deployed in the green environment and tested for functionality and performance. Once the testing results are successful, application traffic is routed from blue to green. Green then becomes the new production.



Canary deployment: a new application code is deployed in a small part of the production infrastructure. Once the application is signed off for release, only a few users are routed to it. This minimizes any impact. With no errors reported, the new version can gradually roll out to the rest of the infrastructure. Consider a way to route new users by exploring several techniques:

- Exposing internal users to the canary deployment before allowing external user access.
- Basing routing on the source IP range.
- Releasing the application in specific geographic regions.

- Using an application logic to unlock new features to specific users and groups. This logic is removed when the application goes live for the rest of the users.

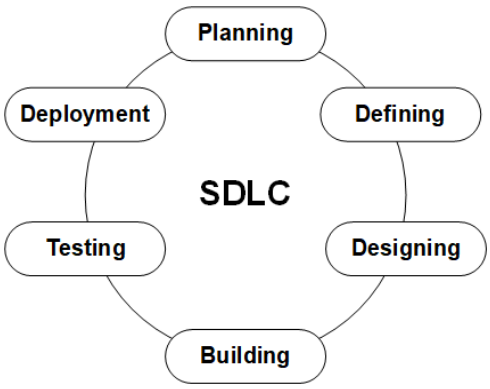


Merging vs rebasing: merging is a non-destructive operation (the existing branches are not changed in any way). On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. The major benefit of rebasing is that you get a much cleaner project history. Also rebasing results in a perfectly linear project history. But, there are two trade-offs for this pristine commit history: safety and traceability. If you don't follow the Golden Rule of Rebasing (never use it on public branches), re-writing project history can be potentially catastrophic for your collaboration workflow. And, less importantly, rebasing loses the context provided by a merge commit.

Force pushing overwrites the remote main branch to match the rebased one from your repository and makes things very confusing for the rest of your team. So, be very careful to use this command only when you know exactly what you're doing.

11. SDLC Methodologies

What is SDLC, its phases and models (predictive vs iterative vs adaptive)?



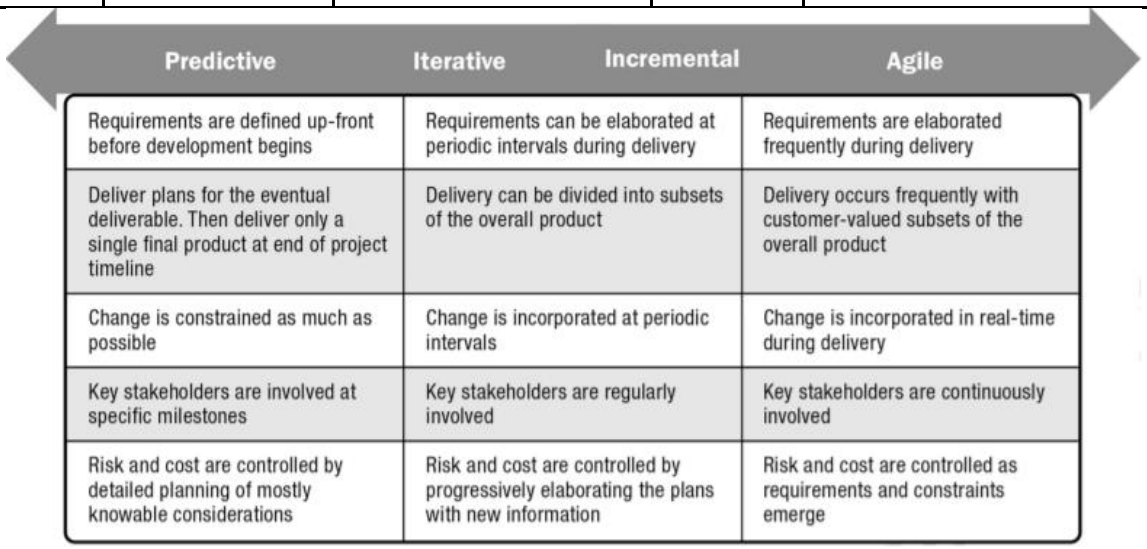
Software Development Life Cycle is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

Stage 1 (planning and requirement analysis) is performed by team's senior members with inputs from the customer, the sales department, market surveys and domain experts in the industry. Then this information is used to plan the basic project approach and to conduct product feasibility study in economical, operational and technical areas. Planning for quality assurance requirements and identification of the risks associated with the project is also done. The outcome of technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2 (defining requirements) is a documentation of product requirements with their following approvement by customer or market analysts. This is done through a Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3 (designing the product architecture) is a process of Design Document Specification's creation. This document includes various approaches of product architecture's building and is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.
Stage 4 (building or developing the product) is a process of programming code's generation as per Design Document Specification of product's building. Code generation can be accomplished without much hassle in case of detailed and organized design. Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code.
Stage 5 (testing the product) refers to testing of the product where defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the Software Requirement Specification (testing activities are mostly involved in all stages of SDLC).
Stage 6 (deployment in market and maintenance) is a formal release in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment. Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.
There are various software development life cycle models defined and designed which are followed during the software development process.
6 effective SDLC models: Which one is best? (Waterfall, V-Shaped, Iterative, Spiral, Big Bang, Agile)
Predictive vs. Adaptive SDLC: What is the Difference?

Model	Requirements	Activities	Delivery	Goal
Predictive	Fixed	Performed once for entire project	Single delivery	Management of cost
Iterative	Dynamic	Repeated until correct		Correctness of solution
Incremental	Dynamic	Performed once for a given increment	Frequent smaller deliveries	Speed
Agile	Dynamic	Repeated until correct		Customer value via frequent deliveries and feedback



Predictive	Iterative	Incremental	Agile
Requirements are defined up-front before development begins	Requirements can be elaborated at periodic intervals during delivery		Requirements are elaborated frequently during delivery
Deliver plans for the eventual deliverable. Then deliver only a single final product at end of project timeline	Delivery can be divided into subsets of the overall product		Delivery occurs frequently with customer-valued subsets of the overall product
Change is constrained as much as possible	Change is incorporated at periodic intervals		Change is incorporated in real-time during delivery
Key stakeholders are involved at specific milestones	Key stakeholders are regularly involved		Key stakeholders are continuously involved
Risk and cost are controlled by detailed planning of mostly knowable considerations	Risk and cost are controlled by progressively elaborating the plans with new information		Risk and cost are controlled as requirements and constraints emerge

Waterfall – stages, pros/cons

- SDLC - Waterfall Model
- What is Waterfall Model in SDLC? Advantages and Disadvantages

Agile explanation, manifesto, pros/cons

Essence of agile consists in four suggested ways of working, outlined in the Agile Manifesto. Agile's values focus on being adaptable and collaborative, prioritizing people over processes and using "working software" to get products market-ready as quickly as possible.

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

What Are the Advantages and Disadvantages of Agile and Scrum?

SCRUM, roles, ceremonies: The 2020 Scrum Guide TM

SCRUM – how to handle a non-trivial scenario (i.e. senior manager asks directly to do something → only product owner can set up tasks, goals; critical issue before vacation → check a process with PM, file a ticket, investigate, update the ticket with findings, delegate to the most experienced guy in that area).

Scrum teams are by definition cross-functional. How do you understand this term?

According to Scrum Guide, a cross-functional team is organized around a product, a defined portion of a product, a service, or a customer value stream, and must include all competencies needed to accomplish their work without depending on others that are not part of the team. These teams deliver products iteratively and incrementally, maximizing opportunities for feedback and ensuring a potentially useful version of working product is always available. Additionally, these teams should be self-organizing, choosing how best to accomplish their work rather than being directed by others outside the team. Quite simply, your teams need to hold within themselves all of the skills required to plan and deliver the fastest realization of value for a new product or change request, without having to pass their work to others for next steps. This would include code writing, user story elaboration, and manual and automated product testing.

Do you have grooming sessions? What is their format and outcome?

7 Backlog Grooming Tips for Beginners

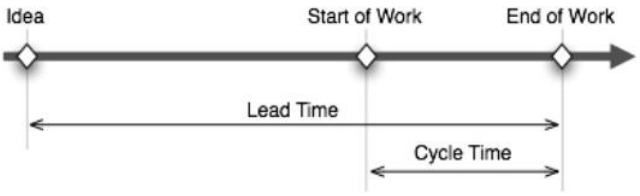
Kanban explanation, pros/cons, metrics

A kanban board is a physical or virtual board designed to help visualize work, limit work-in-progress, and maximize efficiency. It consists of columns designated to some stages of the kanban cycle and tasks located floating through columns. The most straightforward boards have requested, in progress and done columns.

Pros: **alignment** (everyone are kept on the same page), **exposing chokepoints and bottlenecks** (when columns start getting too long, they indicate a disconnect between resources and demand), **no hard-and-fast rules** (true flexibility regarding prioritization, decision-making, what processes are necessary to enter or exit each stage), **flexibility** (queue of work to be done can be rearranged and reprioritized based on new data and strategic inputs without impacting the rest of the product development workflow).

Cons: **timeless** (there's no way to know if an item is one day or three months away from exiting to the next column within a particular stage); **potentially outdated** (if you don't update the board, it's not accurately reflecting things).

Lead time is the period between a new task's appearance in your workflow and its final departure from the system.
Cycle time is the total amount of elapsed time between task's start and finish.



Throughput is the total amount of work delivered in a certain time period (number of tasks).
WIP is the actual work-in-progress time.

Cycle Time = WIP / Throughput

What to Measure and Why: Kanban Metrics

User story DoD (definition of done) and DoR (definition of ready)

What are DoD and DoR in Scrum? The DoD is usually a short document in the form of a checklist, that defines when a product backlog item (i.e. user story) is considered "done". The best way to check whether something is "done" is to simply ship it! A typical DoD might look like this example:

- Automated tests are written and all tests are green
- Code is refactored and reviewed
- Code is integrated with master branch
- Deployed to staging environment
- Translated into English and German

DoD is a checklist of what needs to be done to a product backlog item before the team can start implementing it. Note that the DoR is NOT part of the Scrum Guide – and that is for good reason. The DoR should not be used as a phase gate for Sprint Planning or as a way to push away responsibility. A typical DoR might look like this example:

- PO and Dev Team need to have talked about the story at least once
- Story must have clear business value
- Effort needs to be estimated
- Story must be broken down enough to fit a single sprint
- Story needs at least one acceptance criteria

A user story should be independent, negotiable, valuable, estimable, small and testable.

Extreme Programming best practices

Extreme Programming (XP): Values, Principles, and Practices

Key feature – emphasis on technical aspects of software development
When to use: highly-adaptive development, risky projects, small teams, automated testing, readiness to accept new culture and knowledge, customer participation

12. Estimations

Estimation techniques (SP, T-Shirt, planning poker, top-bottom, bottom-up, PERT)

Estimates are an essential starting point for any project and agile teams rely on them as well. Agile estimation is all about refining product backlog items into smaller implementable items and then estimating what it takes to convert that backlog item into a done item (not for coding and unit testing alone). Agile estimates are provided by the development team members, they are not imposed by the managers. The type of estimation traditionally used by product managers is absolute unit estimation. (number of time units like hours, days, or weeks are used). However, estimation in relative units is a more effective and popular approach nowadays. Absolute unit estimation can vary significantly based on individual skills and experience. Relative estimation leads to consensus much quicker because regardless of their skills or experience level, developers find it easier to agree that one item is twice as big as another item. Agile teams commonly use two series of numbers for ranking users' stories or work items relatively, the **Fibonacci** and **exponential** series. The Fibonacci series starts with a 0, followed

by one and then each subsequent number is the sum of the two previous numbers. While the actual series continues infinitely, for ranking purposes, the number is capped at 13. So, values **0, 1, 2, 3, 5, 8, 13** are used to rank the difficulty and scope of product backlog items. For the exponential series, each number after the first two is two times the previous number, so the exponential values are **0, 1, 2, 4, 8, 16**. Each series also has an infinity value estimate (item is too big to be estimated). Teams also use a question mark for items that they cannot estimate because they don't have enough information. Items with a relative estimate of 0 are items that the team has already done or something that is so insignificant that the team can finish it very quickly (this estimate is optional). If you look at Fibonacci series or exponential series, you will notice that as items get bigger, the difference between two consecutive numbers also gets bigger. This is aligned with the fact that as work items get bigger and more complicated, the uncertainty associated with the implementation increases. Usage of T-shirt sizes **(Extra Small [XS], Small [S], Medium [M], Large [L], Extra Large [XL])** is one more way to think of features' relative sizes. This is an even greater departure from the numeric system, and like all good gross-level estimation units can in no way be associated with a specific length of time. Other arbitrary tokens of measurement include Gummi Bears, NUTS (Nebulous Units of Time), foot-pounds. Teams may create their own estimation units with a bit of fun in doing so. **Main agile metrics: sprint burndown, epic and release burndown, velocity, control chart, cumulative flow diagram.**

- What are story points and how do you estimate them?
- Five agile metrics you won't hate | Atlassian
- What is a Gantt chart? | Atlassian
- Agile estimation techniques
- A Complete Guide To Project Estimation Techniques
- How to Estimate Projects: Top 5 Project Estimation Techniques for Small Businesses
- How to do effective project estimation during Pre-Sales?

Story points pros:

- constant while adjusting the amount of points promised in the sprint
- relative sizing of estimation is easier to do than trying to anticipate exactly how long a task is going to take
- easier to commit to points than hours from a psychological perspective
- less chance for task to be grown in hours

Story point cons:

- imprecise by nature
- work better with stable team (not gaining or losing members frequently)
- takes a while to determine a real velocity of team
- easy for misunderstanding and misuse of story points (there is a learning curve, which can cause confusion)

Hours pros:

- more precise than points (doesn't mean that more accurate)
- easier to handle teams that gain and lose team members
- project managers and businesses often prefer to work for hours
- easier to understand

Hours cons:

- can be perceived as "when will this be done?" which is not strictly correct
- task can be easily grown
- dependency on developer's individual skills and experience
- need to get detailed in order to determine hours (this slows a process of estimation

How to Prevent Estimate Inflation
IT project risk management: risk examples, categories, prevention, mitigation
Unknown complexity and estimation
Dealing with unclear requirements
Why IT project estimates fail and what to do about it

Estimation of multipliers (communication, bug fixing, additional effort on accessibility, tests, cross-browser support, responsive/adaptive design, localization/globalization) usually depends

on process's, stakeholders' and developers' peculiarities. In general, all these risks should be taken into account during the planning and grooming sessions but it's not uncommon that some auxiliary tasks can be created later because of changed features' description.

13. Team Management and Soft Skills

Task delegation – what can be and what can't be delegated, how to delegate, levels of delegation. Difference between delegation and assignment.

Delegation mistakes

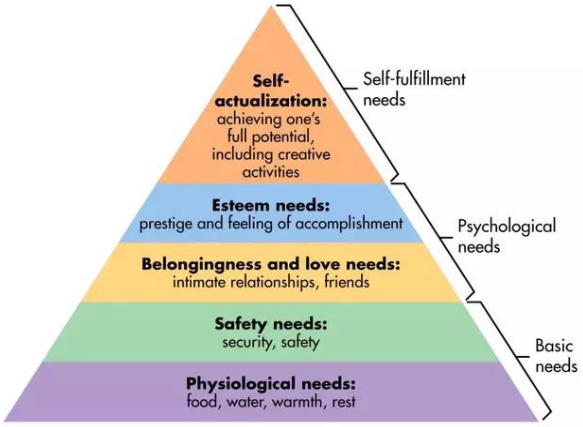
1. Not able to differentiate between delegating and training.
2. Providing vague instructions.
3. Picking the wrong person.
4. Delegating a task without any following monitoring.
5. Expecting perfection.
6. Not sharing the rewards and credit.
7. Not knowing what to delegate.

What can be delegated: low priority task, which is something that doesn't come under your core focus area; less important tasks that eat away at your time and energy; laying the groundwork such as collecting resources, prospect research, data entry, etc; tasks that you are not good at doing; tasks that your teammates can do better; something that you want your team to learn.

- Successful Delegation - Team Management Training from MindTools.com
- How to Delegate Tasks Effectively (and Why It's Important)
- 5 Levels of Remarkably Effective Delegation | Inc.com
- What Is the Difference Between Assignment and Delegation? (assignment is a process of transferring responsibility and accountability; delegation is a process by which responsibility and authority for performing a task or activity is transferred to another person)

How to decompose and assign tasks, how to manage situations when less senior teammates want to get more complicated tasks (i.e. assign a part of task, control execution, create a separate task to get acquainted with a new technology)? Decomposition of business stories into tasks depends on the experience and skills of each team member. In general, all stories should be divided into simple use cases which can be easily implemented in programming code. If a junior dev wants to get more complicated tasks, he/she should express this wish during the planning/grooming. In case of some non-trivial tasks a more senior dev should provide a list of key points (under each task) that junior dev should adhere for successful realization.

How to convince dev to do boring tasks: advice to automate when possible; advice to look at another angle, use a new approach, technology and learn from it; demonstrate justice (rotate tasks across developers fairly in accordance with their skills and wishes).



Motivation techniques – how to motivate team members and mentee

- 7 Powerful ways to motivate your team
- 9 Super Effective Ways to Motivate Your Team | Inc.com

Time management

The Ultimate List: 58 Time Management Techniques

The Pomodoro Technique is a time management system that encourages people to work with the time they have – rather than against it.

- Break your workday into 25-minute chunks separated by 5-minute breaks.
- These intervals are referred to as pomodoros.
- After about 4 pomodoros, you take a longer break of about 15 to 20 minutes.

Eisenhower's Urgent/Important Principle

1. **Important and urgent** (ones that you could not have foreseen, and others that you've left until the last minute; leave some time in your schedule to handle unexpected issues and unplanned important activities).
2. **Important but not urgent** (these are the activities that help you achieve your personal and professional goals, and complete important work).
3. **Not important but urgent** (are things that prevent you from achieving your goals, ask yourself whether you can reschedule or delegate them).
4. **Not important and not urgent** (you can simply ignore or cancel many of them).

How to handle conflicts inside the team

Conflicts, How to resolve conflicts inside a team

Deal with low employee performance

Article: 5 smart ways to deal with low performers — People Matters

1. Improve your feedback (let low performer know that you notice)
2. Prepare an action plan (without a viable solution, any negative feedback is useless)
3. Look at the communication channels (team members need to know exactly what's expected of them)
4. Set performance goals
5. Follow up on implementation (ensure that the proposed changes are being implemented)

How To Deal With Poor Employee Performance - eLearning Industry

Tips for Effectively Managing Software Developers | Pluralsight

Onboarding process and its goals. Best practices of onboarding for newcomers. Onboarding documentation.

A One-Stop Guide To Remote Software Developers Onboarding

Onboarding process

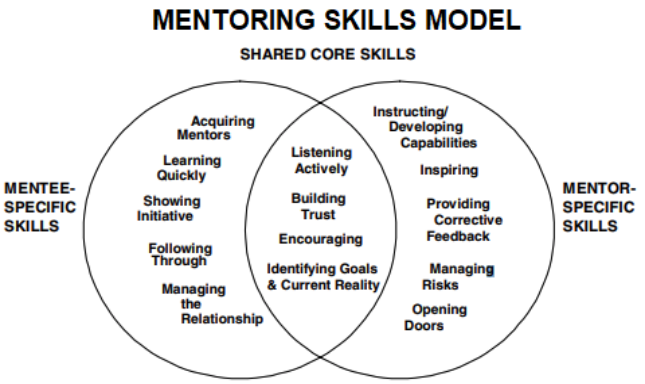
1. Collect new employee contact information and personal data.
2. Request for an initial access for the new employee and provide him/her with a list of requirable accesses (personal account and similar) to retrieve on his/her own.
3. Introduce he/she to a team, make acquaintance with staff in charge, add to chats, meetings and mailing lists.
4. Give an overview of the company, team and/or project.
5. Assign a mentor for a new employee.
6. Provide an entry point to a project wiki, onboarding program or any.
7. Set up the nearest tasks and midterm plan.
8. Schedule trainings.

All stuff like project's description, architecture, modules, non-obvious solutions, installation process, CI/CD pipelines, branching strategies, environment(s), description of how to contribute and how to use the project should be in project wiki.

How to develop and mentor other team members

[Developers mentoring other developers: practices I've seen work well]

[A Guide for Mentors]



Mentoring process

1. Building the relationship
2. Exchanging information and setting goals
3. Working towards goals or deepening the engagement.
4. Ending the formal mentoring relationship and planning for the future.

Mentoring Best Practices

- Think of yourself as a “learning facilitator” rather than a person with all answers. Help your protégé find people and resources that go beyond your experience and wisdom on a topic.
- Emphasize questions over advice giving. Use probes that help your protégé think more broadly and deeply. If he/she talks only about facts, ask about feelings. If he/she focuses on feelings, ask him/her to review the facts. If he/she seems stuck in an immediate crisis, help him/her see a big picture.
- When requested, share your own experiences, lessons learned, and advice. Emphasize how your experiences could be different from his or her experiences and are merely examples. Limit your urge to solve the problem for him or her.
- Resist the temptation to control the relationship and steer its outcomes; your protégé is responsible for his or her own growth.
- Help your protégé see alternative interpretations and approaches.
- Build your protégé’s confidence through supportive feedback.
- Encourage, inspire, and challenge your protégé to achieve his or her goals.
- Help your protégé reflect on successful strategies he or she has used in the past that could apply to new challenges.
- Be spontaneous now and then. Beyond your planned conversations, call or email “out of the blue” just to leave an encouraging word or piece of new information.
- Reflect on your mentoring practice. Request feedback.
- Enjoy the privilege of mentoring. Know that your efforts will likely have a significant impact on your protégé’s development as well as your own.

Effective meetings techniques

1. Decide whether you really need a meeting.
2. Schedule it at the right time.
3. Create an agenda.
4. Get input for your plan.
5. Read through materials and hash out issues beforehand.
6. Make it a stand-up if possible.
7. Come together with a ritual.
8. Stick to the agenda.
9. Embrace tabling.
10. Ditch long meetings for frequent meetings (timeframe).
11. Make it a safe place for input.
12. Start with expectations.
13. Clarify your decision-making procedure.

14. Close with action items and communicate them clearly.
15. Solicit feedback for the next meeting.

Always be evaluating your meetings. Consider your own thoughts and those of your attendees and make sure that, to the best of your ability, each meeting that you run is efficient, engaging, purposeful and productive.

Negotiations techniques

5 Good Negotiation Techniques - PON

1. Reframe anxiety as excitement;
2. Anchor the discussion with a draft agreement.
3. Draw on the power of silence.
4. Ask for advice. Professional negotiators often assume that asking the other party for advice will convey weakness, inexperience, or both.
5. Put a fair offer to the test with final-offer arbitration.

10 Negotiation Techniques to Make You

1. Prepare, prepare, prepare.
2. Pay attention to timing
3. Leave behind your ego.
4. Ramp up your listening skills.
5. If you don't ask, you don't get.
6. Anticipate compromise.
7. Offer and expect commitment.
8. Don't absorb their problems.
9. Stick to your principles.
10. Close with confirmation.

Own recent failure and the way it was managed/handled

- Interview Question: "How Do You Handle Failure"
- Interview Question: "Tell Me About a Time You Failed"

Difference between D2 and D3 developers: **D2** actively learns and adopts the technology and tools defined by the team; enforces the team processes, making sure everybody understands the benefits and tradeoffs; makes an impact on one or more subsystems or team pods. **D3** is go-to person for one or more technologies and takes initiative to learn new ones; challenges the team processes, looking for ways to improve them; makes an impact on the whole team, not just on specific parts of it.

How do you interview candidate and define level

How to Conduct an Effective Interview, How to conduct a structured interview