What is Cosmos:

The Cosmos-SDK is an framework for building:

- multi-asset public Proof-of-Stake (PoS) blockchains, like the Cosmos Hub
- permissioned Proof-Of-Authority (PoA) blockchains
- Allow developers to easily create custom blockchains from scratch that can natively inter-operate with other blockchains

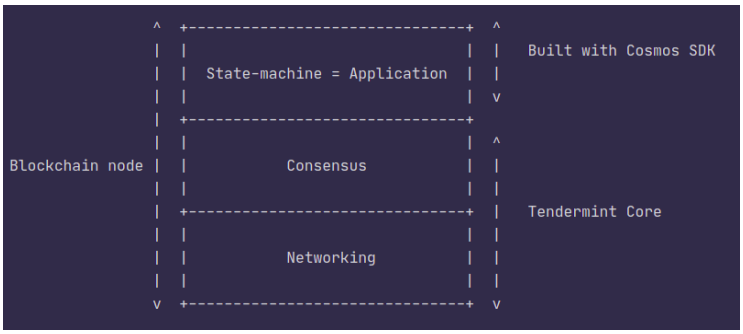
by combining open_source modules. Integrating modules is as simple as *importing* them into your blockchain application

Blockchains built with the Cosmos SDK are generally referred to as ***application-specific blockchains***.

The SDK as the npm-like framework to build secure blockchain applications on top of *Tendermint.*

What are application-specific blockchains:

Instead of building a decentralized application on top of an underlying blockchain like Ethereum, developers build their own blockchain from the ground up. This means building a full-node client, a light-client, and all the necessary interfaces (CLI, REST, ...) to interact with the nodes. Their state-machine incorporates a virtual-machine that is able to interpret turing-complete programs called Smart contracts. Smart contracts are developed with specific programming languages that can be interpreted by the underlying virtual-machine. EVM does not allow developers to implement automatic execution of code



- One development paradigm in the blockchain world today is that of ***virtual-machine blockchains*** like Ethereum, where development generally revolves around building a decentralized applications on top of an existing blockchain as a set of *smart contracts*.
- smart contract are good for single-use applications (e.g. ICOs – initial token offering) but not good for more complex applications
- smart contracts can be limiting in terms of flexibility, sovereignty and performance.
- An application-specific blockchain only operates a single application, so that the application does not compete with others for computation and storage. This is the opposite of most non-sharded virtual-machine blockchains today, where smart contracts all compete for computation and storage.
- An application-specific blockchain is a blockchain customized to operate a single application: developers have all the freedom to make the design decisions required for the application to run optimally.
- They can also provide better sovereignty, security and performance.

Why the Cosmos SDK?

- The default consensus engine available within the SDK is Tendermint, BFT consensus engine. It is gold standard consensus engine for building *Proof-of-Stake* systems.
    - **What is practical Byzantine Fault Tolerance?**

      Practical Byzantine Fault Tolerance is a system that has a primary node and secondary nodes. These nodes work together to reach a consensus.

      Here's a basic breakdown of how practical Byzantine Fault Tolerance works:

      - The client makes a request to the primary node.
      - The primary node sends that request on to the secondary nodes.
      - The nodes process the request, provide the service, and respond to the client.
      - The client waits until it has received the same response from *m*+1 nodes, with *m* being the maximum number of faulty/malicious nodes the system allows.

      In a practical BFT system, the maximum number of faulty/malicious nodes can't be equal to or greater than one-third of the system's total nodes.
- Easy to build blockchains out of composable modules.
- The SDK is inspired by capabilities-based security
- C*osmos Hub*, *IRIS Hub*, ***Binance Chain***, *Terra* or Kava are building on the Cosmos SDK.

Blockchain Architecture:

At its core, a blockchain is a **replicated deterministic state machine**.

- A state machine is a computer science concept whereby a machine can have multiple states, but only one at any given time. There is a state, which describes the current state of the system, and **transactions**, that trigger state transitions. In practice, the **transactions are bundled in blocks** to make the process more efficient. Given a state S and a block of transactions B, the state machine will return a new state S'.

How the state-machine is replicated using Tendermint:

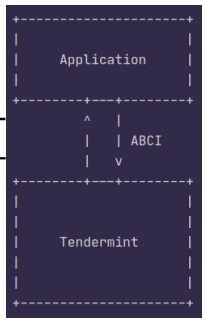
Developers just have to define the state machine by using cosmos SDK, and *Tendermint* will handle *replication* over the network for them.

Tendermint is an engine that is responsible for handling the *networking and consensus layers* of a blockchain. In practice, Tendermint is responsible for *propagating* and *ordering transaction bytes*.

The Tendermint consensus algorithm works with a set of special nodes called Validators. Validators are responsible for adding blocks of transactions to the blockchain. At any given block, there is a validator set V. **A validator in V is chosen by the algorithm to be the proposer of the next block**. This block is considered valid if more than two thirds of V signed a prevote and a precommit on it, and if all the transactions that it contains are valid. **The validator set can be changed by rules written in the state-machine.**

ABCI:

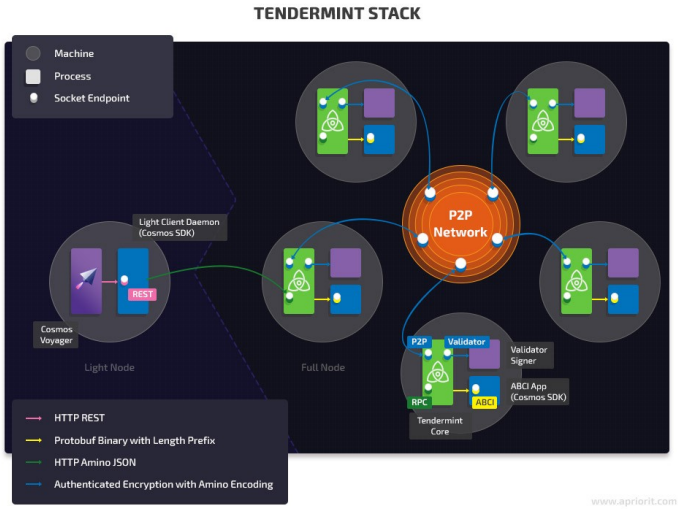
Tendermint passes transactions to the application through an interface called the **ABCI, which the application must implement.**



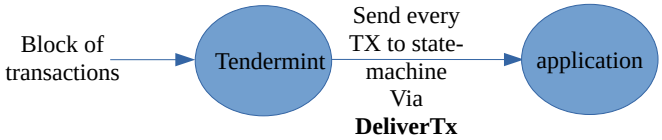
Tendermint only handles transaction **bytes**. It has no knowledge of what these bytes mean. All Tendermint does is order these transaction bytes deterministically(?). Tendermint passes the bytes to the application via the ABCI, and **expects a return code to inform it if the messages** contained in the transactions were successfully processed or not.

CheckTx is used to protect the mempool of full-nodes (primary nodes mentioned in BFT steps) against spam transactions. A special handler called the **AnteHandler** is used to execute a series of validation steps such as checking for sufficient fees and validating the signatures. If the checks are valid, the transaction is added to the *mempool* and relayed to peer nodes (secondary nodes in BFT steps).

- Full nodes are located in tendermint engine.
- Hence, mempool is in the tendermint as well.
- ABCI is an interface in the tendermint.
- At SDK core, the SDK is a boilerplate implementation of the ABCI in Golang.



DeliverTx When a valid block is received by Tendermint Core, each transaction in the block is passed to the application via DeliverTx in order to be processed. It is during this stage that the *state transitions* occur. The **AnteHandler** executes again along with the actual *Msg service* RPC for each message in the transaction.



BeginBlock/EndBlock: These messages are executed at the beginning and the end of each block, whether the block contains transaction or not. It is useful to trigger *automatic execution* of logic.

Main Components of the Cosmos SDK:

At its core, the SDK is a boilerplate (It's a code that can be used by many applications/contexts with little or no change-s needed by a programming language very often all around the programs you write in that language) implementation of the ABCI in Golang. It comes with a **multistore** to persist data and a **router** to handle transactions.

how transactions are handled by an application built on top of the Cosmos SDK when transferred from Tendermint via DeliverTx:

- ***Decode transactions*** received from the Tendermint consensus engine (remember that Tendermint only deals with []bytes).
- ***Extract messages*** from transactions and do basic sanity ***checks***.
- ***Route*** each message to the appropriate module so that it can be processed.
- Commit ***state*** changes.

Baseapp:

baseapp is the boilerplate implementation of a Cosmos SDK application. It comes with an implementation of the ABCI to handle the connection with the underlying consensus engine(?).

The goal of baseapp is to provide a “secure interface between the store and the extensible state machine” while defining as little about the state machine as possible (staying true to the ABCI).

The custom application defined in **app.go** is an extension of baseapp. When a transaction is relayed by Tendermint to the application, app uses baseapp's methods to route them to the appropriate module. baseapp implements most of the core logic for the application, including all the ABCI methods and the routing logic.

Multistore:

For persisting state in KVStores. These KVStores only accept the **[]byte** type as value and therefore any custom structure needs to be marshalled using a **codec** before being stored. The multistore (a store of stores) abstraction is used to **divide the state in distinct compartments**, each managed by its own module

Modules:

SDK applications are built by aggregating a collection of interoperable modules. Each module can be seen as a little state-machine.

Each module defines a subset of the state and developers need to define the subset of the state handled by the module, as well as custom message types that modify the state (Note: messages are extracted from transactions by baseapp)

Each module contains its own message/transaction processor, while the SDK is responsible for routing each message to its respective module.

Security is based on object-capabilities. In practice, this means that instead of having each module keep an access control list for other modules, each module implements special objects called keepers that can be passed to other modules to grant a pre-defined set of capabilities. (functionality only can be called by the methods of keeper object in each module)

- x/auth: Used to manage accounts and signatures.
- x/bank: Used to enable tokens and token transfers.
- x/staking + x/slashing: Used to build Proof-Of-Stake blockchains.

Start Creating an application

The first thing defined in “**app.go**” is the type of the application.

```
// SimApp extends an ABCI application, but with most of its parameters exported.
// They are exported for convenience in creating helper functions, as object
// capabilities aren't needed for testing.
type SimApp struct {
        *baseapp.BaseApp
        legacyAmino       *codec.LegacyAmino
        appCodec          codec.Marshaler
        interfaceRegistry types.InterfaceRegistry

        invCheckPeriod uint

        // keys to access the substores
        keys    map[string]*sdk.KVStoreKey
        tkeys   map[string]*sdk.TransientStoreKey
        memKeys map[string]*sdk.MemoryStoreKey

        // keepers
        AccountKeeper    authkeeper.AccountKeeper
        BankKeeper       bankkeeper.Keeper
        CapabilityKeeper *capabilitykeeper.Keeper
        StakingKeeper    stakingkeeper.Keeper
        SlashingKeeper   slashingkeeper.Keeper
        MintKeeper       mintkeeper.Keeper
        DistrKeeper      distrkeeper.Keeper
        GovKeeper        govkeeper.Keeper
        CrisisKeeper     crisiskeeper.Keeper
        UpgradeKeeper    upgradekeeper.Keeper
        ParamsKeeper     paramskeeper.Keeper
        IBCKeeper        *ibckeeper.Keeper // IBC Keeper must be a pointer in the app,
        EvidenceKeeper   evidencekeeper.Keeper
        TransferKeeper   ibctransferkeeper.Keeper

        // make scoped keepers public for test purposes
        ScopedIBCKeeper      capabilitykeeper.ScopedKeeper
        ScopedTransferKeeper capabilitykeeper.ScopedKeeper
        ScopedIBCMockKeeper  capabilitykeeper.ScopedKeeper

        // the module manager
        mm *module.Manager

        // simulation manager
        sm *module.SimulationManager
}
```

Then we call **constructor function** to construct a new application of the type that defined in the above. It must fulfill the AppCreator signature in order to be used in the start command of the application's daemon command.

```
// AppCreator is a function that allows us to lazily initialize an
// application using various configurations.
AppCreator func(log.Logger, dbm.DB, io.Writer, AppOptions) Application
```

Here’s an example of application constructor from simapp type mentioned before.

In the application's constructor, the **SetOrderInitGenesis** has to be called before the **SetInitChainer** because the order in which the modules' InitGenesis calls is of matter.

The SetOrderBeginBlock and SetOrderEndBlock methods have to be called before the SetBeginBlocker and SetEndBlocker functions.

```
// NewSimApp returns a reference to an initialized SimApp.
func NewSimApp(
	logger log.Logger, db dbm.DB, traceStore io.Writer, loadLatest bool, skipUpgradeHe
	homePath string, invCheckPeriod uint, encodingConfig simappparams.EncodingConfig,
	appOpts servertypes.AppOptions, baseAppOptions ...func(*baseapp.BaseApp),
) *SimApp {

	// TODO: Remove cdc in favor of appCodec once all modules are migrated.
	appCodec := encodingConfig.Marshaler
	legacyAmino := encodingConfig.Amino
	interfaceRegistry := encodingConfig.InterfaceRegistry

	bApp := baseapp.NewBaseApp(appName, logger, db, encodingConfig.TxConfig.TxDecoder(
	bApp.SetCommitMultiStoreTracer(traceStore)
	bApp.SetAppVersion(version.Version)
	bApp.SetInterfaceRegistry(interfaceRegistry)

	keys := sdk.NewKVStoreKeys(
		authtypes.StoreKey, banktypes.StoreKey, stakingtypes.StoreKey,
		minttypes.StoreKey, distrtypes.StoreKey, slashingtypes.StoreKey,
		govtypes.StoreKey, paramstypes.StoreKey, ibchost.StoreKey, upgradetypes.St
		evidencetypes.StoreKey, ibctransfertypes.StoreKey, capabilitytypes.StoreKe
	)
	tkeys := sdk.NewTransientStoreKeys(paramstypes.TStoreKey)
	memKeys := sdk.NewMemoryStoreKeys(capabilitytypes.MemStoreKey)

	app := &SimApp{
		BaseApp:           bApp,
		legacyAmino:       legacyAmino,
		appCodec:          appCodec,
		interfaceRegistry: interfaceRegistry,
		invCheckPeriod:    invCheckPeriod,
		keys:              keys,
		tkeys:             tkeys,
		memKeys:           memKeys,
	}
```

When the node is started at appBlockHeight == 0 (i.e. on genesis), application will receive the InitChain messages from Tendermint. Then the function **InitChainer** is called and initializes the state from the genesis file. InitChainer calls the **InitGenesis** function of the module manager, which in turn will call the InitGenesis function of each of the modules it contains.

An example of an InitChainer from simapp:

```
// InitChainer application update at chain initialization
func (app *SimApp) InitChainer(ctx sdk.Context, req abci.RequestInitChain) abci.ResponseInit
	var genesisState GenesisState
	if err := tmjson.Unmarshal(req.AppStateBytes, &genesisState); err != nil {
		panic(err)
	}
	return app.mm.InitGenesis(ctx, app.appCodec, genesisState)
}
```

The SDK offers developers the possibility to implement automatic execution of code as part of their application. This is implemented through two function called BeginBlocker and EndBlocker. They are called when the application receives respectively the BeginBlock and EndBlock messages from the Tendermint engine, which happens at the beginning and at the end of each block.

There is no smart contract on Ethereum with automatic execution. Smart contracts execute as a result of processing transactions. You always need a transaction to trigger/start a function of a contract from external.

Don't introduce non-determinism in **BeginBlocker** or **EndBlocker**. Don't make them too computationally expensive, as gas does not constrain the cost of BeginBlocker and EndBlocker execution.

```
// BeginBlocker application updates every begin block
func (app *SimApp) BeginBlocker(ctx sdk.Context, req abci.RequestBeginBlock) abci.Response
        return app.mm.BeginBlock(ctx, req)
}

// EndBlocker application updates every end block
func (app *SimApp) EndBlocker(ctx sdk.Context, req abci.RequestEndBlock) abci.ResponseEndB
        return app.mm.EndBlock(ctx, req)
}
```

Binary wire **Encoding** of types in the Cosmos SDK can be broken down into two main categories:

- Client encoding mainly revolves around transaction processing and signing
- Store encoding revolves around types used in state-machine transitions and what is ultimately stored in the Merkle tree.

In the codec package, there exists two core interfaces, **Marshaler** and **ProtoMarshaler**. there exists two implementations of Marshaler. The first being AminoCodec, where both binary and JSON serialization is handled via Amino. The second being ProtoCodec, where both binary and JSON serialization is handled via Protobuf.

```
// EncodingConfig specifies the concrete encoding types to use for a given app.
// This is provided for compatibility between protobuf and amino implementations.
type EncodingConfig struct {
        InterfaceRegistry types.InterfaceRegistry
        Marshaler         codec.Marshaler
        TxConfig          client.TxConfig
        Amino             *codec.LegacyAmino
}
```

The InterfaceRegistry is used by the Protobuf codec to handle interfaces that are encoded and decoded (we also say "unpacked") using google.protobuf.Any. ***Any*** could be thought as a struct that contains a type_url (name of a concrete type implementing the interface) and a value (its encoded bytes).

NOTE about "Any" usage:

> sdk.Msgs are encoded using Protobuf Anys. By analyzing each Any's type_url, baseapp's msgServiceRouter routes the sdk.Msg to the corresponding module's Msg service.

Marshaler is the the default codec used throughout the SDK. It is composed of a **BinaryCodec** used to encode and decode *state*, and a **JSONCodec** used to output data to the users.

Application Module Interface:

Modules must implement these interfaces in the ./modules.go file:

AppModuleBasic: implements basic independent elements of the module, such as the codec **AppModule:** handles the bulk of the module methods (including methods that require references to other modules' keepers). These methods are called from the module manager (../building-modules/module-manager.md#manager)

Keeper:

Keepers are the gatekeepers of their module's store(s). To read or write in a module's store and only the module's keeper should hold the key(s) to the module's store(s).

The keeper's constructor function, NewKeeper instantiates a new keeper of the type with a **codec**, **store keys** and potentially **references to other modules' keepers** as parameters. The NewKeeper function is called from the application's constructor. The rest of the file defines the keeper's methods, primarily ***getters*** and ***setters***.

gRPC:

Each module can expose gRPC endpoints, called service methods, and are defined in the module's Protobuf **query.proto** file. A service method is defined by its name, input arguments and output response. The module then needs to:

define a **RegisterGRPCGatewayRoutes** method on AppModuleBasic to wire the client gRPC requests to the correct handler inside the module.

for each service method, define a corresponding **handler**. The handler implements the core logic necessary to serve the gRPC request, and is located in the keeper/grpc_query.go file.

...

Accounts:

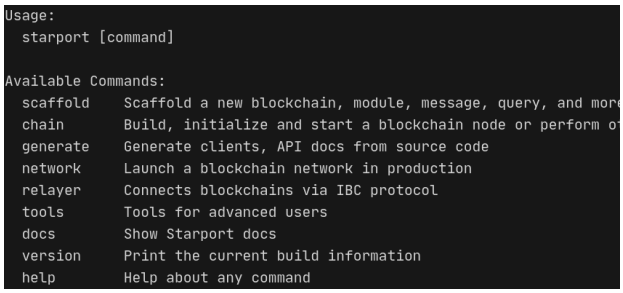
Like most blockchain implementations, Cosmos derives addresses from the public keys. An account is a pair of keys: *publicKey* & *privateKey*. We Asymmetric cryptography for Authentication or Encryption. The former is the one we use in blockchain to sign the transaction and verify if the owner of the account executed that transaction.

HD wallets (Hierarchical-deterministic wallets) uses a single seed (12 or 24 mnemonic) phrase to generate many key pairs to reduce the complexity of key management specially when you work in different blockchains or you may want to transact with others under different aliases. Only the seed phrase needs to be backed up.

Public keys are generally not used to reference accounts (Address does). Public keys do exist and they are accessible through the cryptoTypes.PubKey interface.

Creating your own application using cosmos SDK:

Starport assists with scaffolding modules and integrating them with BaseApp. Starport is a command-line tool that writes code files and updates them when instructed to do so (Ruby on rails for blockchain). Here's the available commands of startport:



After installing the startport we use this command to scaffold a basic chain called "checkers" that you will place under the github path "greenbahar":

>> ***starport scaffold chain github.com/greenbahar/checkers***

To initialize a local testnet with a running node with a single validator we use this command:

>> ***starport chain serve***

This command download/install all dependencies, build protobuf files, compile the application, and add accounts.

Notice: I was going through this tutorial but it wasn't available to proceed further! So I start Doc about the Sdk functionality and keep going on building application later!

Transactions:

They are comprised of metadata that defines a context and one or more sdk.Msg that trigger state changes within a module through the module's Protobuf message service.

In the package type of the SDK we can see different types including:

- Msg
- Fee, FeeTx (Tx is embed in it)
- Signature
- Tx
- TxWithMemo (for platforms like EOS that the transaction must have a memo- e.g. this memo is used for crypto exchanges like binance)
- TxDecoder
- TxEcoder

In general, these types are interfaces and we should create objects (stuct) and implement their methods in order to use them. In this case, transaction objects are types that implement the Tx interface, GetMsgs and ValidateBasic methods.

- GetMsgs: Unwraps the transaction & returns a list of containedsdk.Msg (can be more that one msg)
- ValidateBasic: ABCI message's **CheckTx** and **DeliverTx** use this for stateless checks. Usecases:
  - The **auth module's StdTx**, ValidateBasic function checks that its transactions are signed by the correct number of signers and that the fees do not exceed the user's maximum
  - The sdk.Msg, which perform basic validity checks on messages only. For example:
    - **runTX** first runs ValidateBasic on each message when it checks a transaction created from the auth module.
    - Then it runs the auth module's **AnteHandler**, which calls ValidateBasic for the transaction itself.

Messages:

developers define module messages by adding methods to the *Protobuf Msg service* and defining a MsgServer. Each sdk.Msg is related to exactly one Protobuf Msg service RPC defined inside each module's tx.proto file. A Cosmos SDK app router automatically maps every sdk.Msg to a corresponding RPC service, which routes it to the appropriate method.

Signing Transactions:

Every message in a transaction must be signed by the addresses specified by its GetSigners.

The most used implementation of the Tx interface is the Protobuf Tx message, which is used in SIGN_MODE_DIRECT.

Once signed by all signers, the *BodyBytes*, *AuthInfoBytes*, and *Signatures* are gathered into **TxRaw**, whose serialized bytes are broadcast over the network.

Generating transactions:

To generate a transaction first create a `txBuilder := txConfig.NewTxBuilder()` and add these parameters to it:

- Msgs: The array of messages included in the transaction.
- GasLimit: Option chosen by the users for how to calculate the gas amount they are willing to spend.
- Memo: A note or comment to send with the transaction.
- FeeAmount: The maximum amount the user is willing to pay in fees.
- TimeoutHeight: Block height until which the transaction is valid.
- Signatures: The array of signatures from all signers of the transaction.

```
txBuilder := txConfig.NewTxBuilder()
txBuilder.SetMsgs(...) // and other setters on txBuilder
```

After generating and preparing the transaction you should broadcast it.

```go
func sendTx(ctx context.Context) error {
    // --snip--

    // Create a connection to the gRPC server.
    grpcConn := grpc.Dial(
        "127.0.0.1:9090", // Or your gRPC server address.
        grpc.WithInsecure(), // The Cosmos SDK doesn't support any transport security mechanism.
    )
    defer grpcConn.Close()

    // Broadcast the tx via gRPC. We create a new client for the Protobuf Tx
    // service.
    txClient := tx.NewServiceClient(grpcConn)
    // We then call the BroadcastTx method on this client.
    grpcRes, err := txClient.BroadcastTx(
        ctx,
        &tx.BroadcastTxRequest{
            Mode:    tx.BroadcastMode_BROADCAST_MODE_SYNC,
            TxBytes: txBytes, // Proto-binary of the signed transaction, see previous step.
        },
    )
    if err != nil {
        return err
    }

    fmt.Println(grpcRes.TxResponse.Code) // Should be `0` if the tx is successful

    return nil
}
```

Here https://github.com/cosmos/cosmos-sdk/blob/master/docs/run-node/txs.md#broadcasting-a-transaction-1 is a good document on Generating, Signing and Broadcasting Transactions with the code guide:

keep going through the rest of it...