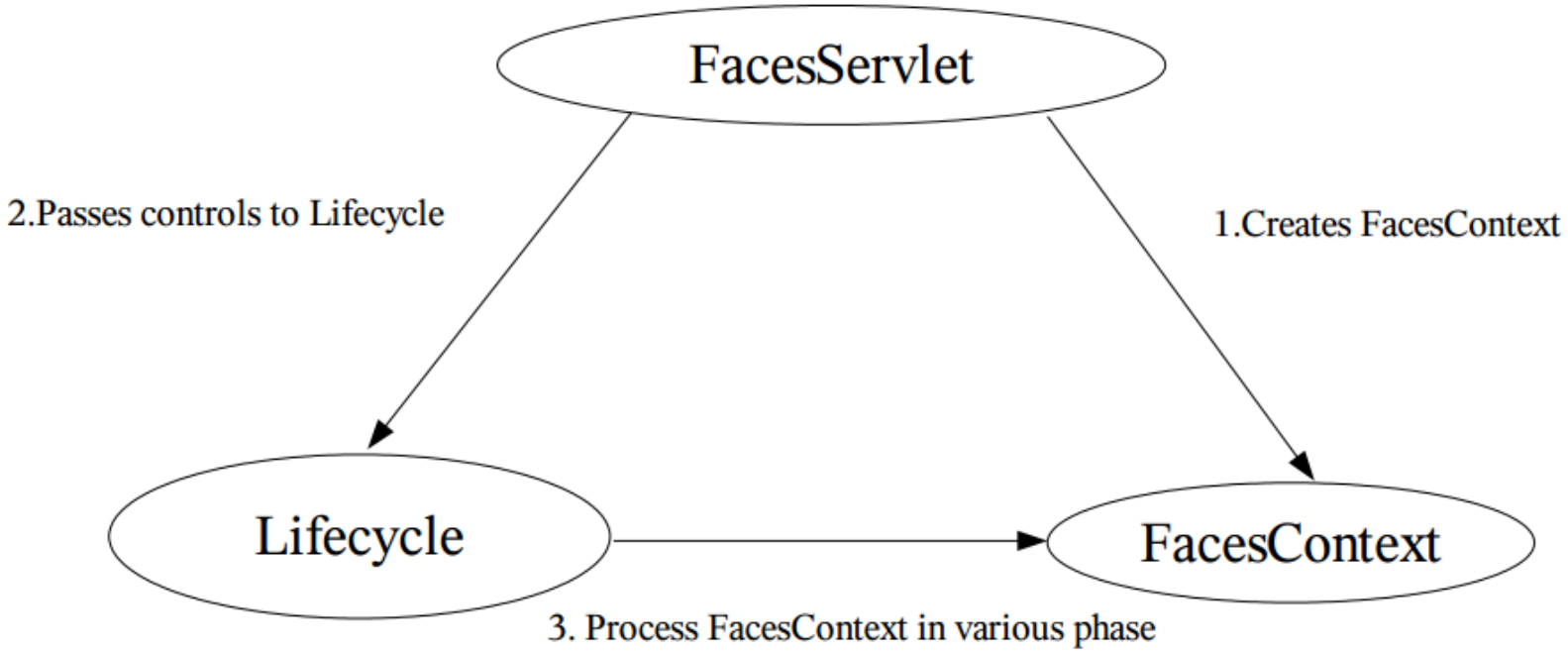
Webfejlesztés előadás anyagok

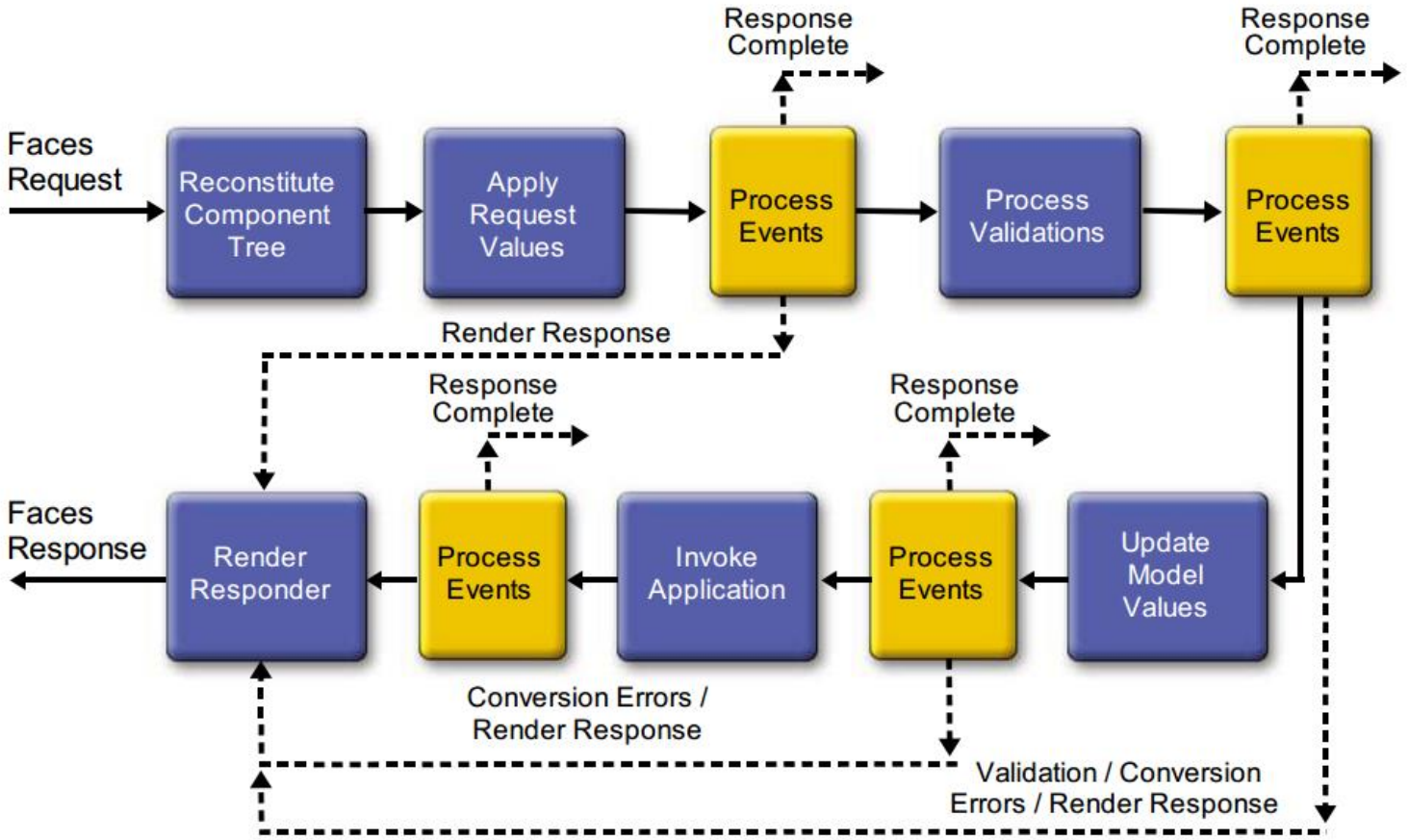
JavaServer™ Faces (JSF) Framework Is…

A server side user interface component framework for Java™ technology-based web applications

Request processing Lifecycle



Request Processing Lifecycle



UI Components

- UIComponent/UIComponentBase

  - Base class for all user interface components

- Standard UIComponent Subclasses

  - UICommand, UIForm, UIOutput
  - UIGraphic, UIInput, UIPanel, UIParameter
  - UISelectBoolean, UISelectMany, UISelectOne

- Example:

```
<h:inputText id="userNo"
             value="#{UserNumberBean.userNumber}"/>
```

Validators and Converters

- Validators - Perform correctness checks on UIInput values
  - Register one or more per component
  - Enqueue one or more messages on errors
  - Standard implementations for common cases
- Converters - Plug-in for conversions:
  - Output: Object to String
  - Input: String to Object
  - Standard implementations for common cases

Rendering Model

Renderers-Adapt components to a specific markup language

- Decoding
- Encoding

RenderKits—Library of Renderers

- Map component classes to component tags
- Is a custom tag library
- Basic HTML RenderKit

Tag	Rendered as
command_button	Login
command_hyperlink	hyperlink

Events and Listeners

- Follows JavaBeans™ Specification design and naming patterns
- Standard events and listeners
  - ActionEvent—UICommand component activated by the user
  - ValueChangedEvent—UIInput component whose value was just changed

Navigation Model

- Application developer responsibility
  - Defined in Application configuration file(Facesconfig.xml)
- Navigation rules
  - Determine which page to go.
  - Navigation case

MVC Architecture

- MVC is the Java-BluePrints-recommended architectural design pattern for interactive applications. MVC separates design concerns, thereby decreasing code duplication, centralizing control, and making the application more extensible. MVC also helps developers with different skill sets focus on their core skills and collaborate through clearly defined interfaces. MVC is the architectural design pattern for the presentation tier.
- MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

Model Layer:

- This is the data layer which consists of the business logic of the system.
- It consists of all the **data** of the application
- It also represents the **state** of the application.
- It consists of **classes** which have the connection to the database.
- The controller connects with model and **fetches the data and sends to the view layer.**
- **The model connects with the database as well and stores the data into a database which is connected to it.**

View Layer:

- This is a **presentation layer**.
- It consists of HTML, JSP, etc. into it.
- It normally **presents the UI** of the application.
- It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.
- This view layer shows the data on UI of the application.

Controller Layer:

- It acts as an **interface between View and Model**.
- It intercepts all the requests which are coming from the view layer.
- It **receives the requests from the view layer and processes the requests** and does the necessary validation for the request.
- This requests is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.

Components of JSF

- User Interface Components
- Events and Listeners
- Page Navigation
- JavaBeans Components
- JSP Pages
- Managed Beans
- Validators
- Convertors
- Server Side Helper Classes
- Application Configuration Resource File

Elements of JSF

- **Describing UI Component:** UI components are the basic reusable components, such as labels, text boxes, used for developing user interfaces. UI components are define as stateful objects maintained on the server side. The server communicates with the clients through these UI components. the components are simple java beans containing properties, methods and events. The JSF UI components are also called Web application user interface components.

- **Describing Renderer:** Most web applications usually send a response to the web browser in the HTML format. Other client devices such as mobile phones or Personal Digital Assistants do not provide HTML browsers. Therefore, web applications need to respond in another markup language. **A JSF Renderer is responsible for two functions, rendering component as HTML markup and processing request attributes.**

- **Describing Validators:** UI components enable a client to interact with the server by giving some data entered by the client to the server. **The data entered by the client must validate for its correctness.** We use java script and java to write the validation logic. Validators can be specified using a component's validator attribute or by nesting JSF provided tags.

- **Describing Backing Beans:** In JSF applications, javaBeans objects handle or store data between the business model and UI components at intermediate stage. These javaBean objects are known as Backing Beans or objects that **handle the interaction between view and model are called backing beans in JSF.**

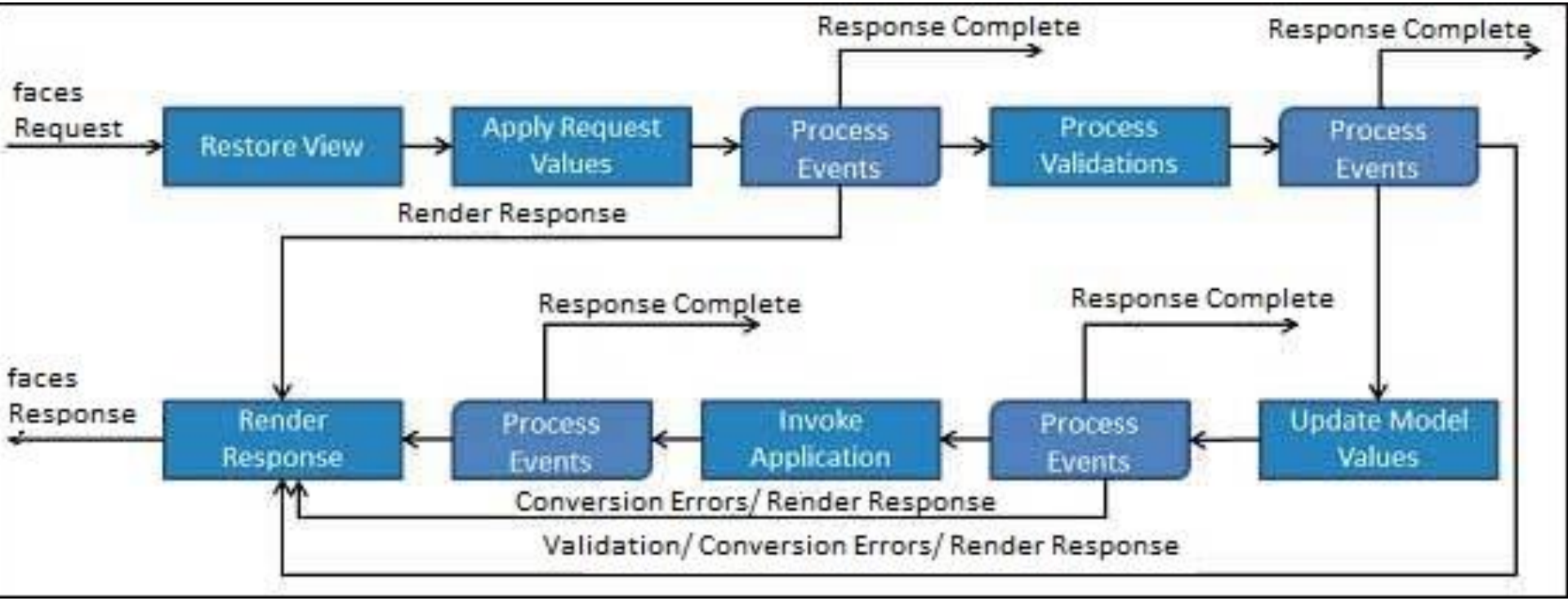
- **Describing Converters:** Web applications interact with clients through the http request and response mechanism. The values entered by the User in all cases will be in string format. Even, when asked to enter numerical values like age or salary whatever is being saved on to the Server end will still be in String format. So, there must be some intermediary which will do the job of **converting the string values into the type that is appropriate for the particular field value.**

- **Describing Events and Listeners:** JSF implements an event driven processing, where events generated on different UI components can be mapped to the execution of some methods. A user can interact with a web application by generating different events on different UI components.

- **Describing Messages:** These are the integral part of the JSF framework. These messages may be associated with some component or cab be used as application level messages
- **Describing Navigation:** A Web application is a **collection of web pages linked together** in a specific order. The clients need to navigate from one page to the other and Web developer needs to implement this navigation logic.

JSF - Life Cycle

- JSF application life cycle consists of six phases which are as follows −
- Restore view phase
- Apply request values phase; process events
- Process validations phase; process events
- Update model values phase; process events
- Invoke application phase; process events
- Render response phase


faces
Request
Restore View
Apply Request
Values
Process
Events
Response Complete
Process
Validations
Process
Events
Response Complete
Render Response
faces
Response
Render
Response
Process
Events
Response Complete
Invoke
Application
Process
Events
Response Complete
Update Model
Values
Conversion Errors/ Render Response
Validation/ Conversion Errors/ Render Response

Phase 1: Restore view

- JSF begins the restore view phase as soon as a link or a button is clicked and JSF receives a request.
- During this phase, JSF builds the view, wires event handlers and validators to UI components and saves the view in the FacesContext instance. The FacesContext instance will now contain all the information required to process a request.

Phase 2: Apply request values

- After the component tree is created/restored, each component in the component tree uses the decode method to extract its new value from the request parameters. Component stores this value. If the conversion fails, an error message is generated and queued on FacesContext. This message will be displayed during the render response phase, along with any validation errors.

Phase 3: Process validation

- During this phase, JSF processes all validators registered on the component tree. It examines the component attribute rules for the validation and compares these rules to the local value stored for the component.
- If the local value is invalid, JSF adds an error message, and the life cycle advances to the render response phase and displays the same page again with the error message.

Phase 4: Update model values

- After the JSF checks that the data is valid, it walks over the component tree and sets the corresponding server-side object properties to the components' local values. JSF will update the bean properties corresponding to the input component's value attribute.

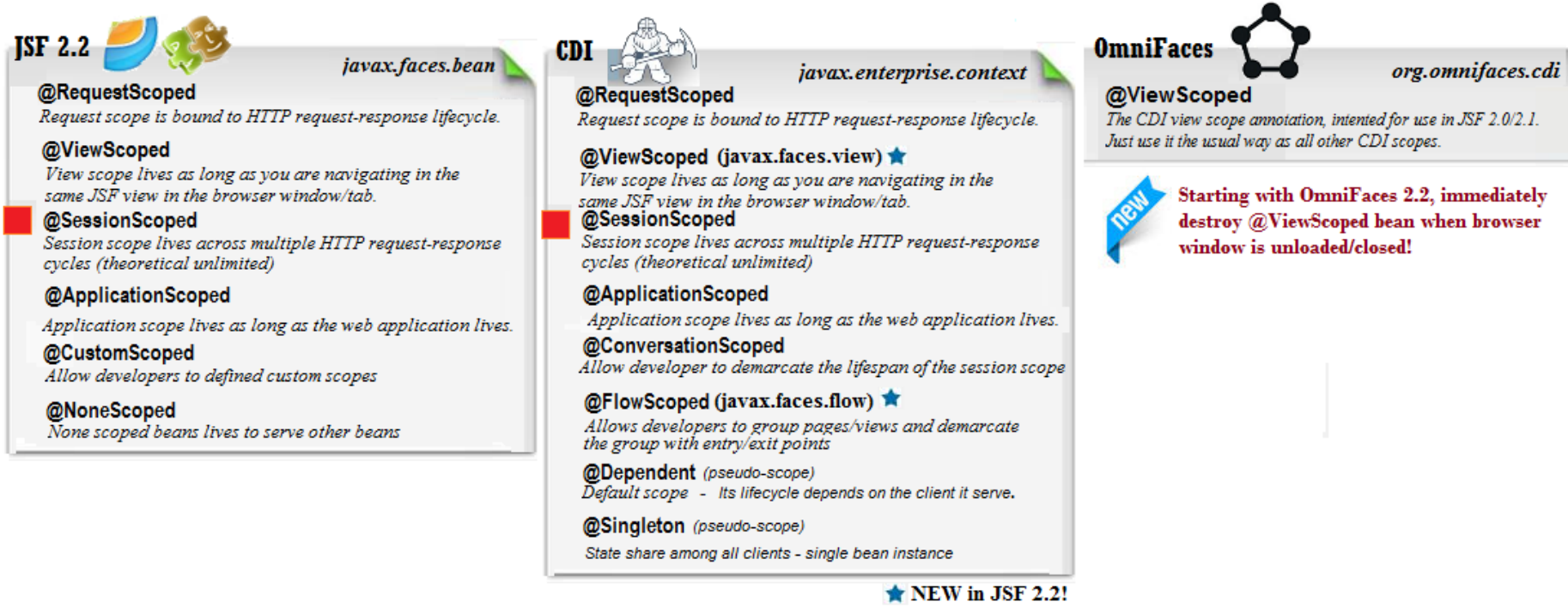
Phase 5: Invoke application

- During this phase, JSF handles any application-level events, such as submitting a form/linking to another page.

Phase 6: Render response

- During this phase, JSF asks container/application server to render the page if the application is using JSP pages. For initial request, the components represented on the page will be added to the component tree as JSP container executes the page.

Bean Scopes


JSF 2.2
javax.faces.bean
@RequestScoped
Request scope is bound to HTTP request-response lifecycle.
@ViewScoped
View scope lives as long as you are navigating in the same JSF view in the browser window/tab.
@SessionScoped
Session scope lives across multiple HTTP request-response cycles (theoretical unlimited)
@ApplicationScoped
Application scope lives as long as the web application lives.
@CustomScoped
Allow developers to defined custom scopes
@NoneScoped
None scoped beans lives to serve other beans
CDI
javax.enterprise.context
@RequestScoped
Request scope is bound to HTTP request-response lifecycle.
@ViewScoped (javax.faces.view)
View scope lives as long as you are navigating in the same JSF view in the browser window/tab.
@SessionScoped
Session scope lives across multiple HTTP request-response cycles (theoretical unlimited)
@ApplicationScoped
Application scope lives as long as the web application lives.
@ConversationScoped
Allow developer to demarcate the lifespan of the session scope
@FlowScoped (javax.faces.flow)
Allows developers to group pages/views and demarcate the group with entry/exit points
@Dependent (pseudo-scope)
Default scope - Its lifecycle depends on the client it serve.
@Singleton (pseudo-scope)
State share among all clients - single bean instance
NEW in JSF 2.2!
OmniFaces
org.omnifaces.cdi
@ViewScoped
The CDI view scope annotation, intented for use in JSF 2.0/2.1. Just use it the usual way as all other CDI scopes.
new
Starting with OmniFaces 2.2, immediately destroy @ViewScoped bean when browser window is unloaded/closed!

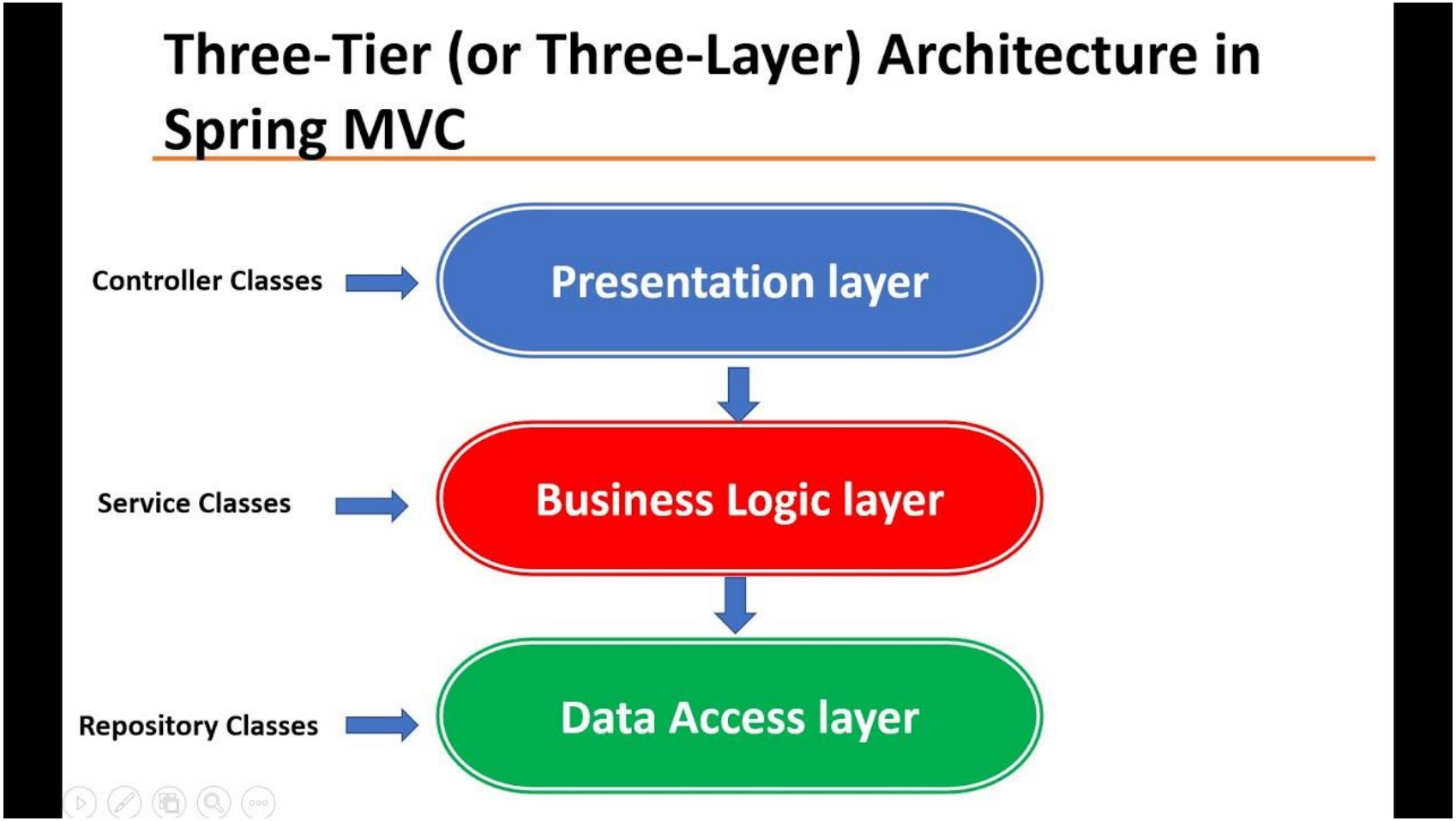
App servers

- Payara
- Tomcat
- Jetty
- Wildfly
- Glassfish
- TomEE
- Websphere
- Weblogic
- NGINX

DB servers

- MySql
- Oracle Sql
- MS Sql
- PostgreSQL
- MongoDB
- IBM DB2
- Redis
- Elasticsearch
- SQLite
- Neo4j

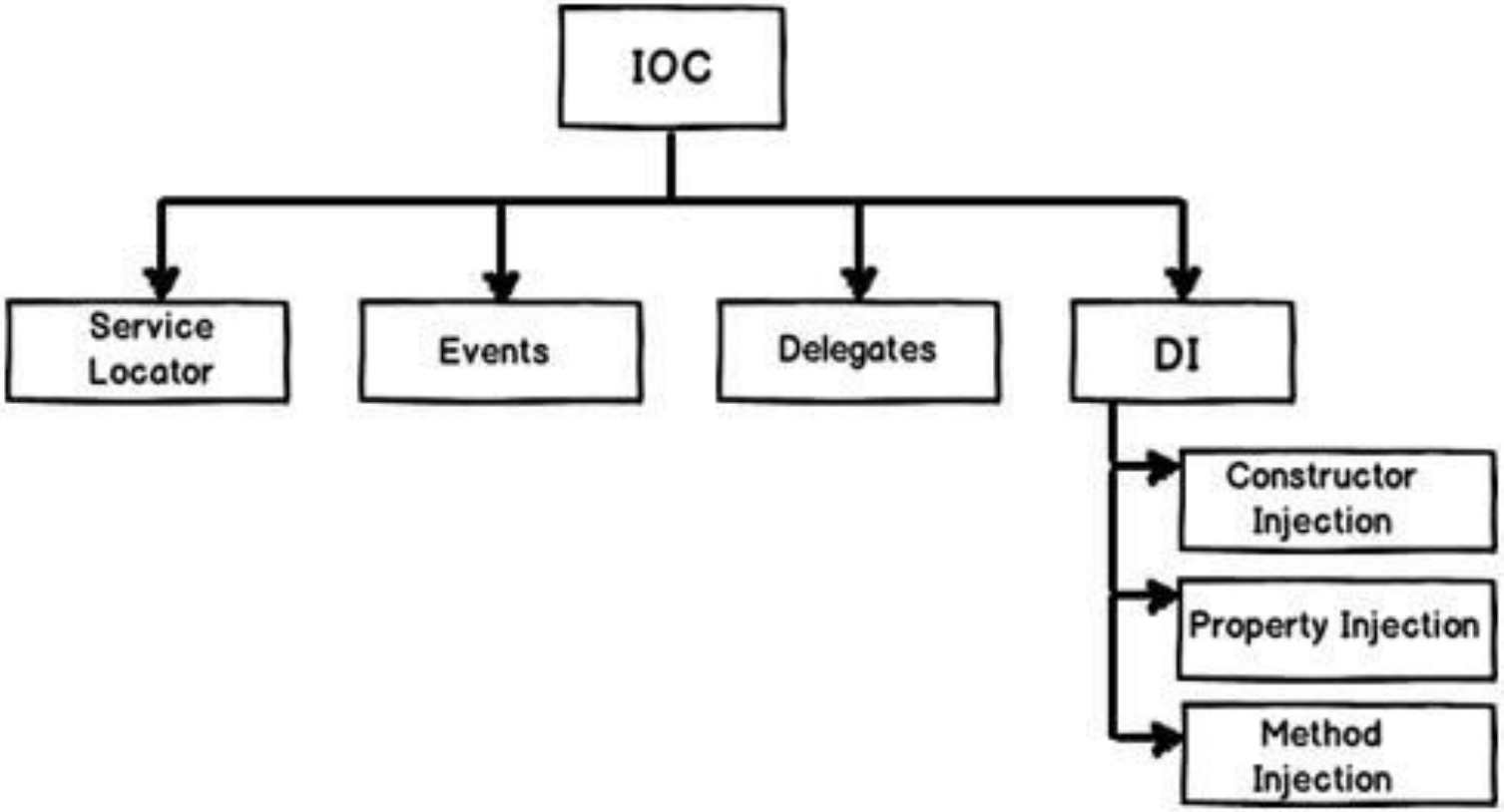
Layered model



Java Bean

- JavaBeans are classes that encapsulate many objects into a single object (the bean). It is a java class that should follow following conventions:
  - Must implement Serializable.
  - It should have a public no-arg constructor.
  - All properties in java bean must be private with public getters and setter methods.

Inversion of Control - IoC



IoC

- Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).

- Dependency Injection (DI) means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.

- Inversion of control is a programming principle that inverts the flow of control in an application. In traditional procedural programming, the code that controls the execution of the program — the main function — instantiates objects, calls methods and even asks the user for input so that the execution can continue and the program can achieve its task. With IoC, it is a framework that does the instantiation, method calls and triggers user actions, having full control of the flow and removing this responsibility from the main function, and by consequence the application.

- Dependency Injection is a software design technique in which the creation and binding of dependencies are done outside of the dependent class. Afterwards, said dependencies are provided already instantiated and ready to be used, hence the term “injection”; in contrast to the dependent class having to instantiate its dependencies internally, and having to know how to configure them, thus causing coupling.

IoC - Spring

←Spring provides Inversion of Control and Dependency Injection

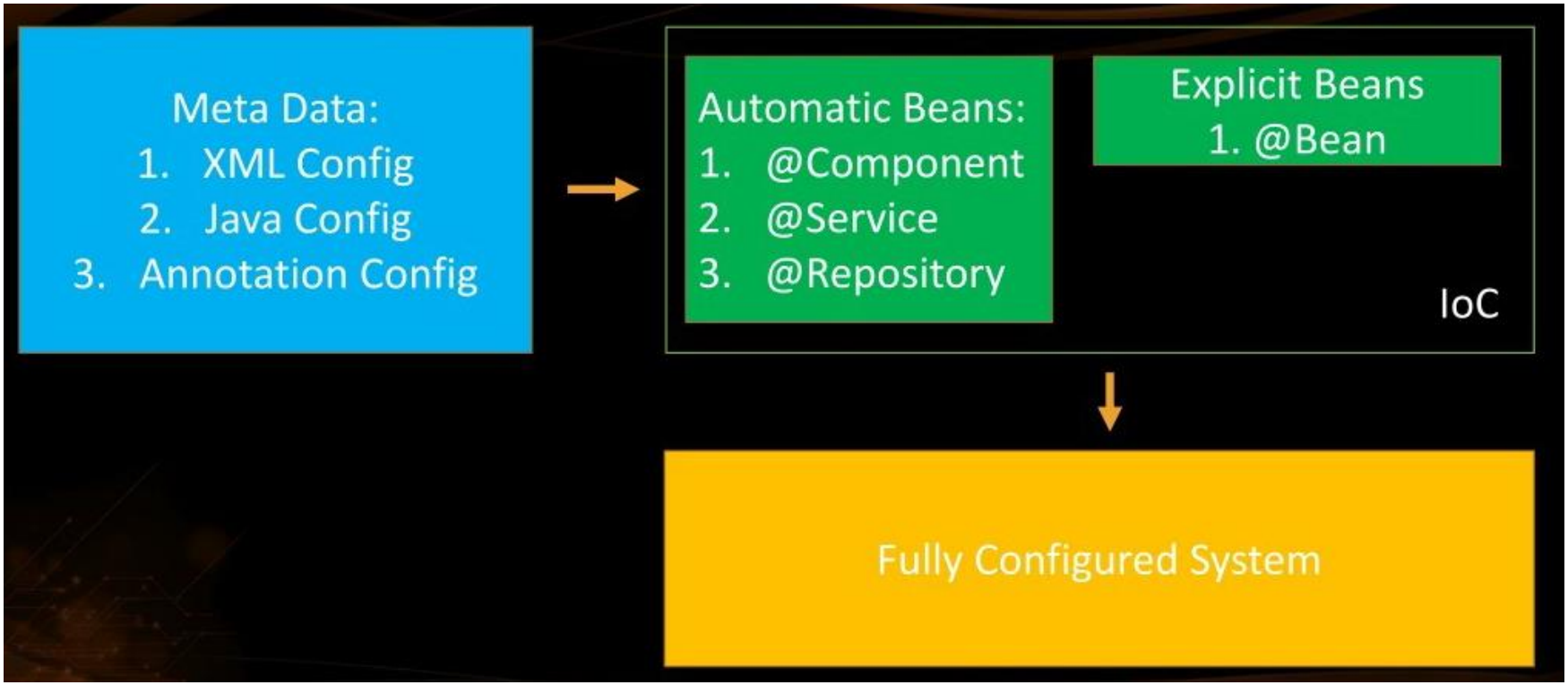
UserServiceImpl.java

```
//Tradiotional Way
public class UserServiceImpl
implements UserService {

private UserRepository
userRepository = new
UserRepository();
}
```

UserServiceImpl.java

```
//Dependency Injection
@Service
public class UserServiceImpl
implements UserService {

@Autowired
private UserRepository
userRepository;
}
```

IoC -Spring


Meta Data:
1. XML Config
2. Java Config
3. Annotation Config
Automatic Beans:
1. @Component
2. @Service
3. @Repository
Explicit Beans
1. @Bean
IoC
Fully Configured System

Spring Beans

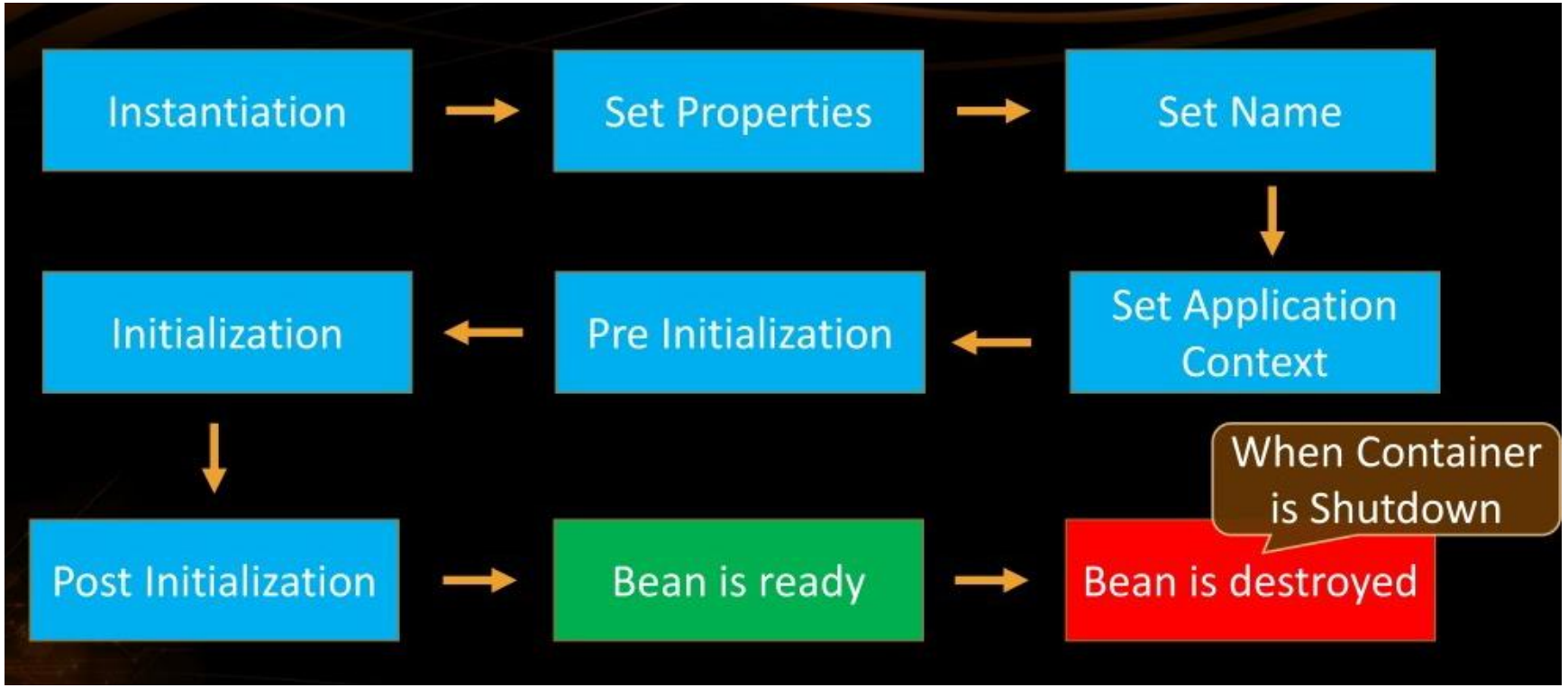
←Object that is instantiated, assembled, and otherwise managed by a Spring IoC container

Dog.java

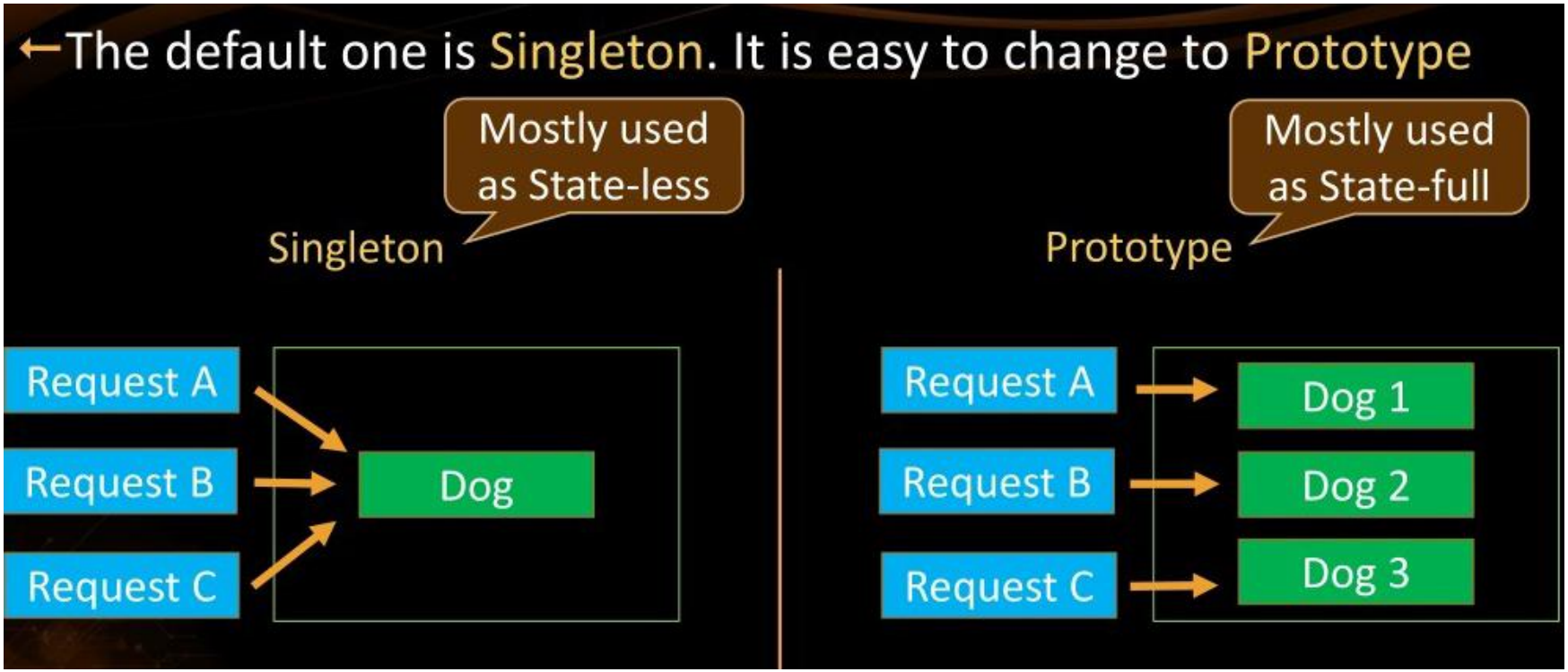
```
public class Dog implements Animal {

    private String name;

    public Dog() {}

    //GETTERS AND SETTERS
}
```

Spring Beans

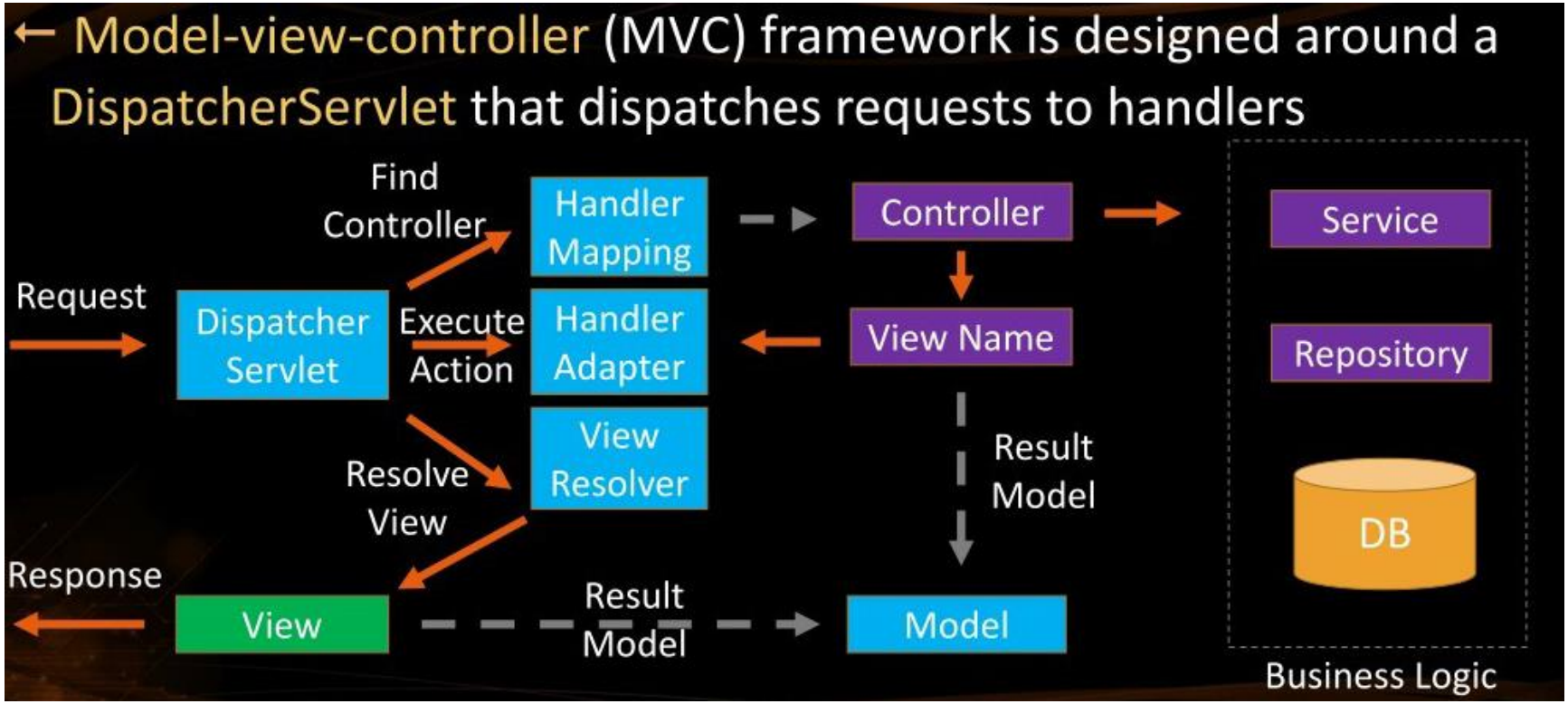
Bean Lifecycle



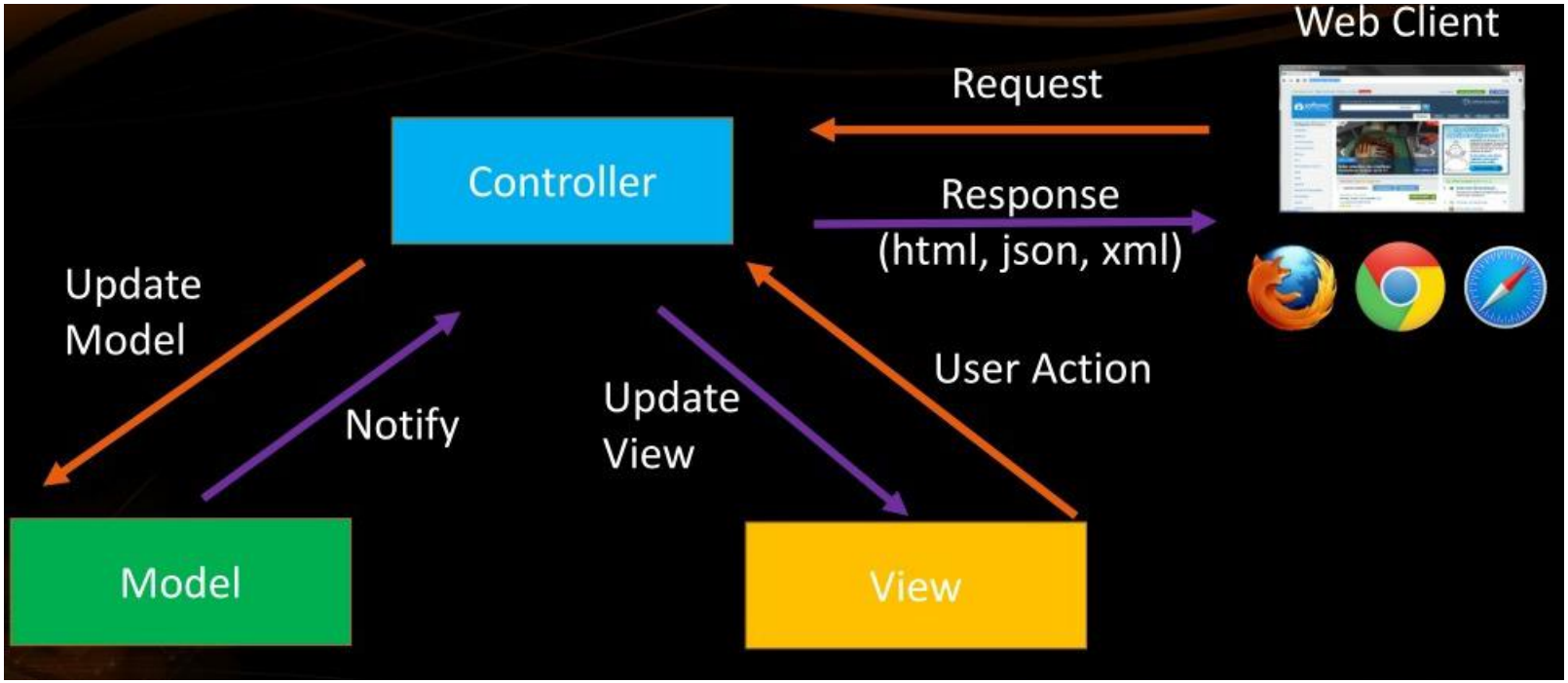
Bean Scope


←The default one is Singleton. It is easy to change to Prototype
Mostly used as State-less
Singleton
Request A
Request B
Request C
Dog
Mostly used as State-full
Prototype
Request A
Request B
Request C
Dog 1
Dog 2
Dog 3

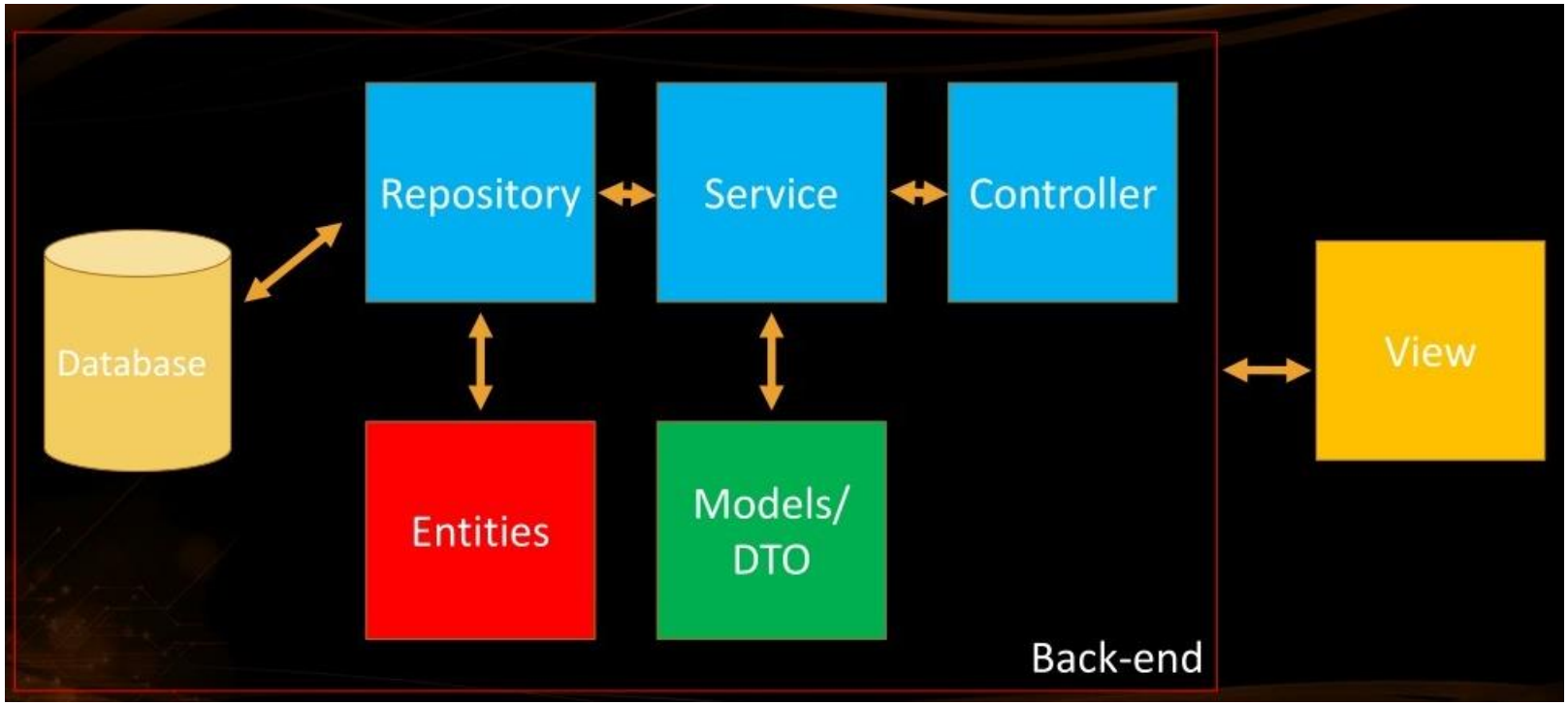
Spring MVC


← Model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers
Find Controller
Handler Mapping
Controller
Service
Request
Dispatcher Servlet
Execute Action
Handler Adapter
View Name
Repository
View Resolver
Result Model
Resolve View
DB
Response
View
Result Model
Model
Business Logic

MVC – Control Flow



Architecture



Entity

←Entity is a lightweight persistence domain object

Cat.java

```
@Entity
@Table(name = "cats")
public class Cat {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private String name;
    //GETTERS AND SETTERS
}
```

Repository


←Persistence layer that works with entities
CatRepository.java
@Repository
public interface CatRepository extends CrudRepository<Cat, Long> {
}

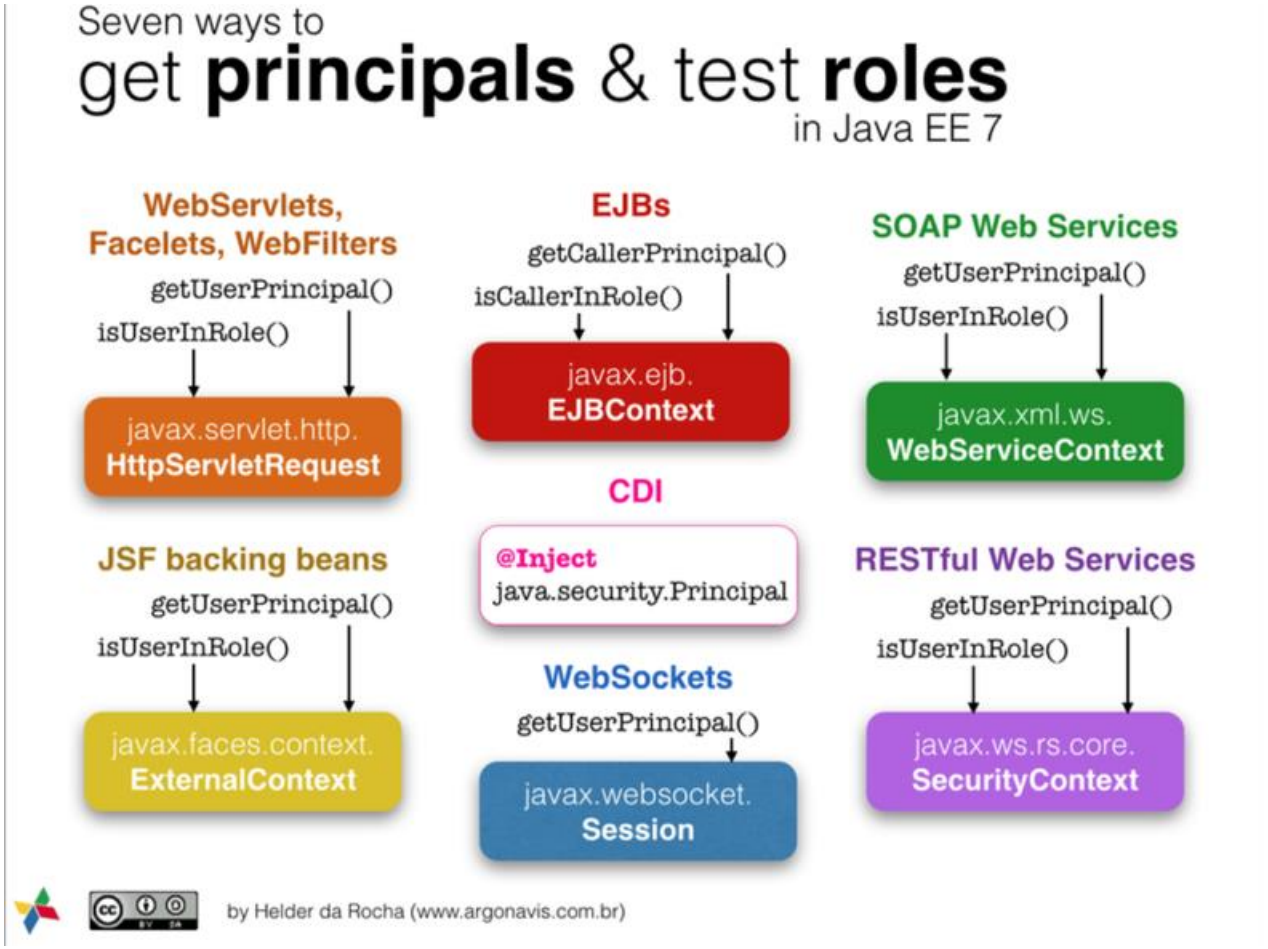
Service

←Business Layer. All the business logic is here.

CatService.java

```
@Service
public class CatServiceImpl implements CatService {

    @Autowired
    private CatRepository catRepository;

    @Override
    public void buyCat(CatModel catModel) {
        //TODO Implement the method
    }
}
```

JEE Security

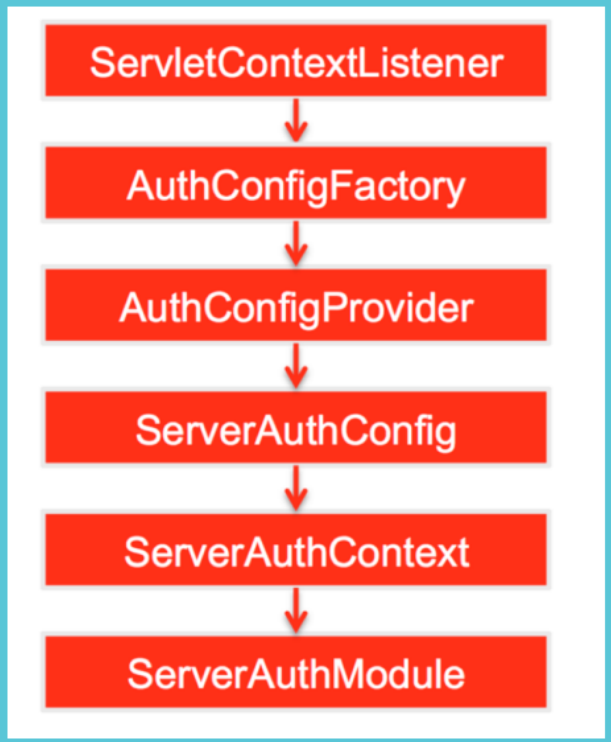


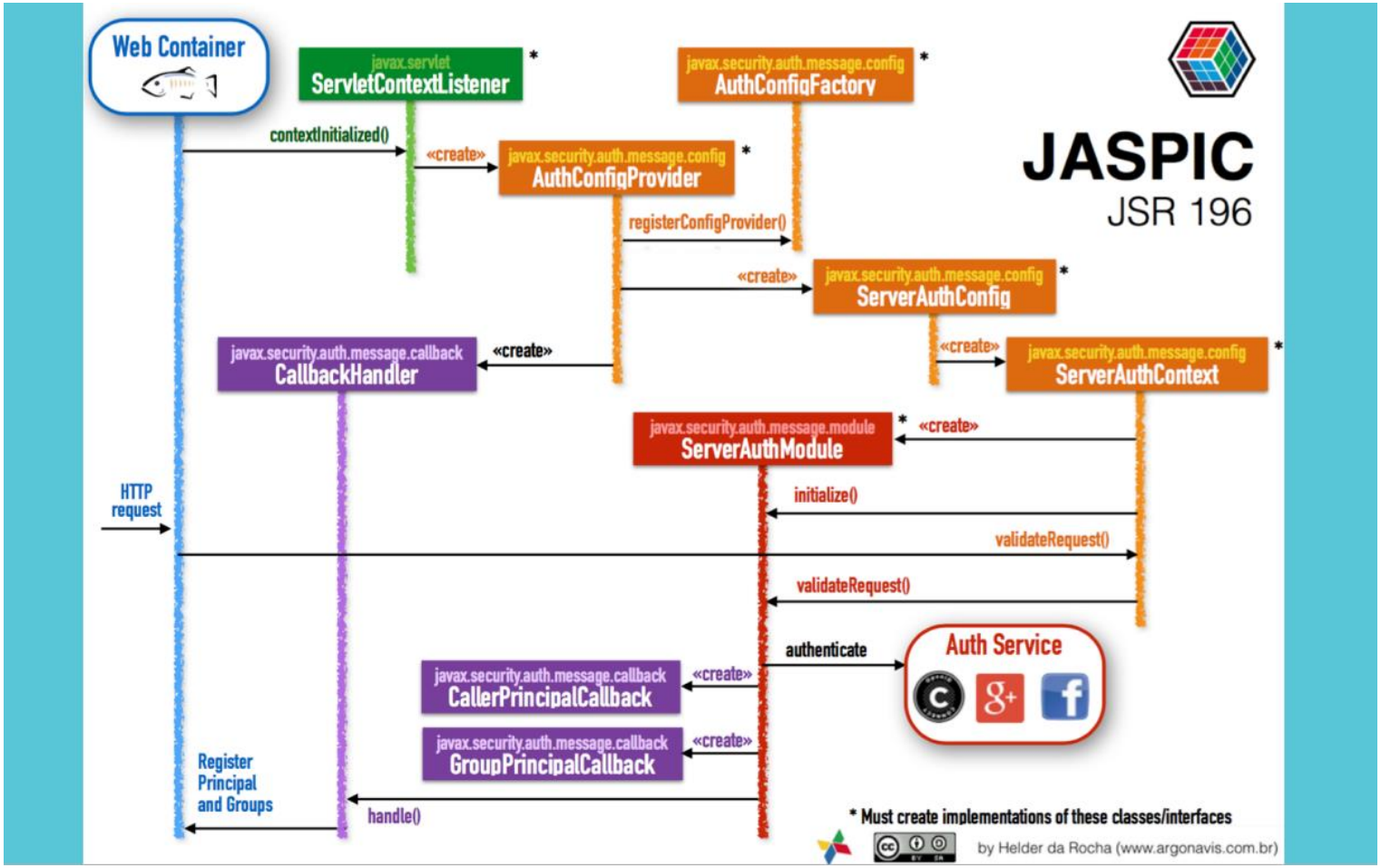
Jaspic

JASPIC

Java Authentication Service Provider Interface for Containers

- Java EE 6
- For custom logic
  - *BASIC/FORM/DIGEST*
- Low Level (per request)
- Verbose




Web Container
javax.servlet
ServletContextListener
*
javax.security.auth.message.config
AuthConfigFactory
*
JASPIC
JSR 196
contextInitialized()
«create»
javax.security.auth.message.config
AuthConfigProvider
*
registerConfigProvider()
«create»
javax.security.auth.message.config
ServerAuthConfig
*
javax.security.auth.message.callback
CallbackHandler
«create»
«create»
javax.security.auth.message.config
ServerAuthContext
*
javax.security.auth.message.module
ServerAuthModule
*
«create»
HTTP
request
initialize()
validateRequest()
validateRequest()
authenticate
Auth Service
javax.security.auth.message.callback
CallerPrincipalCallback
«create»
javax.security.auth.message.callback
GroupPrincipalCallback
«create»
Register
Principal
and Groups
handle()
* Must create implementations of these classes/interfaces
by Helder da Rocha (www.argonavis.com.br)

JSR - 375

JSR-375

- EG discussions started March 2015
- EG Members
  - EE API veterans: many JSRs, many years struggling with Security API
  - 3rd party security framework creators/developers
  - EE platform security implementers
- 10/2016 : EG Updated, switch Spec Lead
- March 13, 2017 : Early Draft Review

JSR - 375

GOALS

- Plug the portability holes
- Modernize
  - Context Dependency Injection (CDI)
  - Intercept at Access Enforcement Points: POJO methods
  - Expression Language (EL)
  - Enable Access Enforcement Points with complex rules
- App Developer Friendly
  - Common security configurations not requiring server changes
  - Annotation defaults not requiring XML

HTTP AUTHENTICATION MECHANISM

- How are credentials retrieved
  - BASIC
  - FORM
    - classic j_security_check, ...
  - CustomForm
    - programmatic
  - Custom
    - For JAX-RS endpoints, ...

IDENTITYSTORE

- Verify credentials
  - LDAP
  - DATABASE
    - with configurable queries
  - EMBEDDED
    - Easy for testing with hardcoded values
  - Custom
    - Whatever your need is

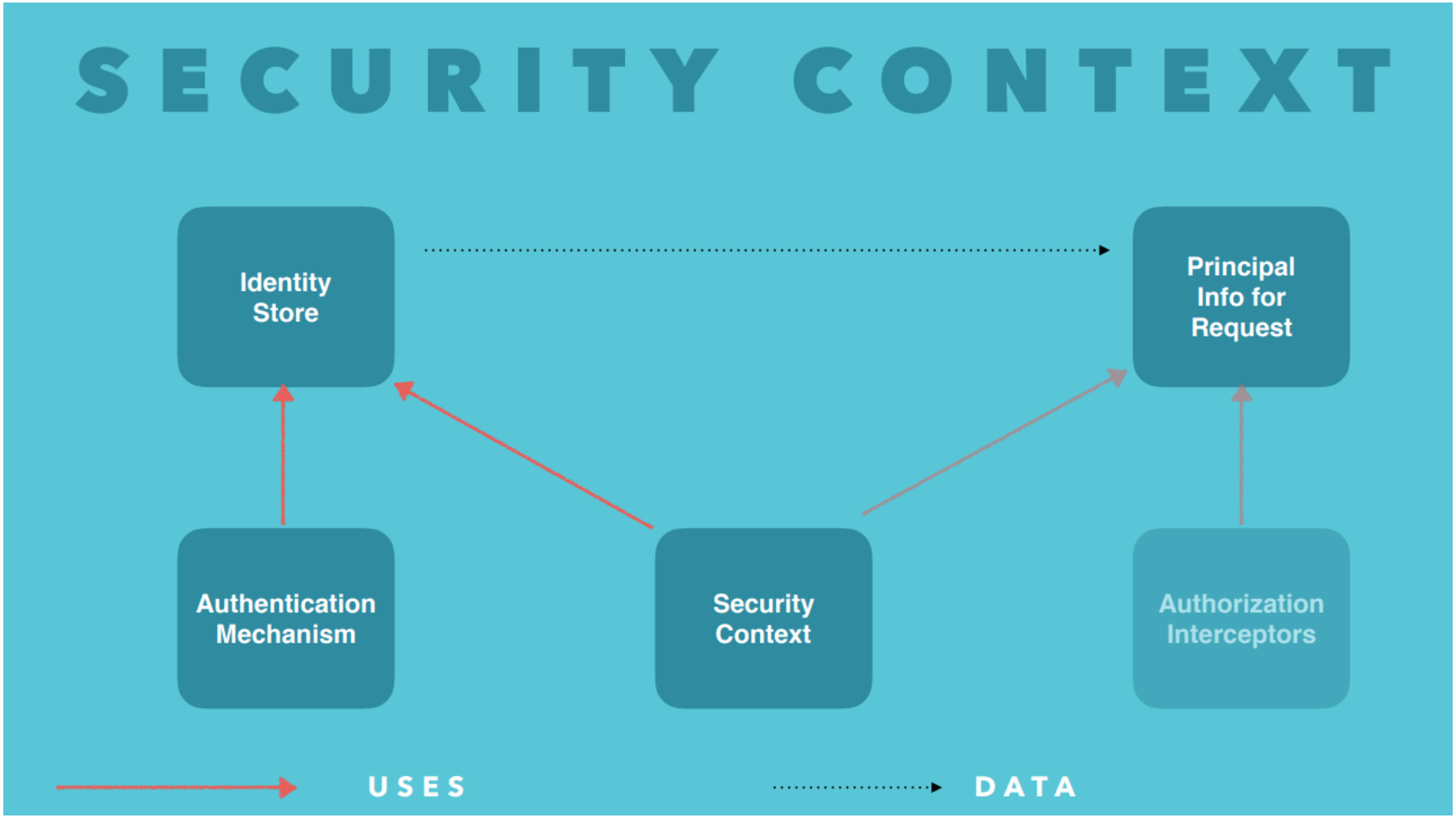
TRIPLE A

- Authentication
  - Verifying that a user is who she says she is.
- Authorisation
  - He can execute the allowed actions within their privilege.
- Accounting
  - Audit

IDENTITY STORE HANDLER

- IdentityStoreHandler
  - Handles multiple defined Identity Stores

- ValidationType on IdentityStore
  - BOTH
  - AUTHENTICATION
  - AUTHORIZATION


SECURITY CONTEXT
Identity Store
Principal Info for Request
Authentication Mechanism
Security Context
Authorization Interceptors
USES
DATA

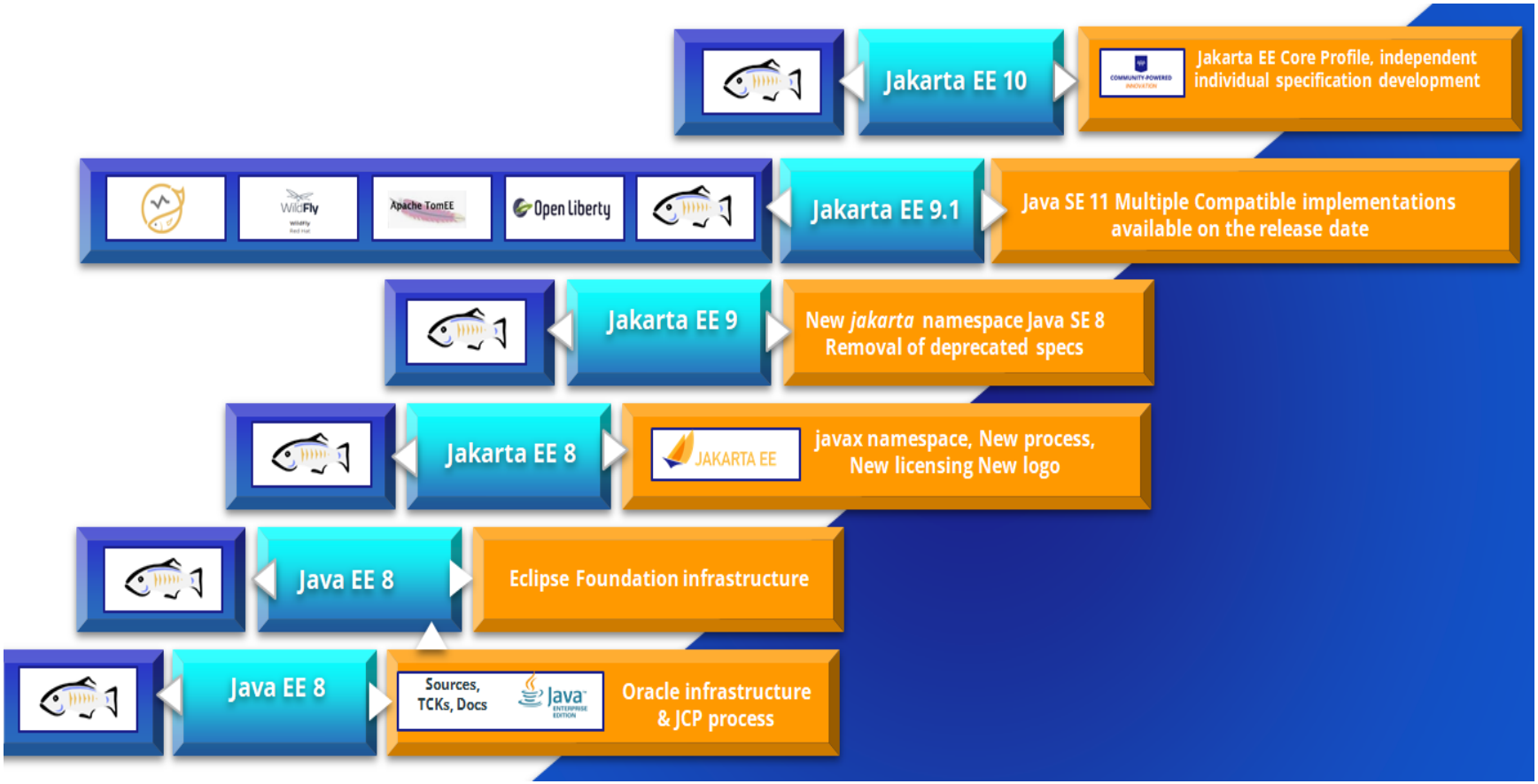
JEE Security Terminology

- Authentication Mechanism The mechanism by which authentication is performed. This mechanism interacts with the caller to obtain credentials and invokes an identity store to match the given credentials with a known user (identity). If a match is found, the Authentication Mechanism uses the found identity to populate attributes (principals) to build an authenticated Subject. If a match is not found, the Authentication Mechanism reports a failed authentication, the caller is not logged in, and is unable to be given authorization.
- Caller, Caller Principal A caller is a user that is making a request to an application, or invoking an application API. A Caller Principal is a Principal object representing that user. This specification uses the term caller in preference to the term user in most contexts.
- HAM Abbreviation for HttpAuthenticationMechanism, an interface defined by this specification. Identity Store An Identity Store is a component that can access application-specific security data such as users, groups, roles, and permissions. It can be thought of as a security-specific DAO (Data Access Object). Synonyms: security provider, repository, store, login module (JAAS), identity manager, service provider, relying party, authenticator, user service.
- Identity Stores usually have a 1-to-1 correlation with a data source such as a relational database, LDAP directory, file system, or other similar resource. As such, implementations of the IdentityStore interface use data source-specific APIs to discover authorization data (roles, permissions, etc), such as JDBC, File IO, Hibernate or JPA, or any other Data Access API.
- JACC JSR-115, "Java Authorization Contract for Containers", version 1.5 [JACC].
- JASPIC JSR-196, "Java Authentication SPI for Containers", version 1.1 [JASPIC].

Jakarta

- In September 2017, Java EE technologies moved to the Eclipse Foundation, where they now continue to evolve under the Jakarta EE brand
- javax namespace -> jakarta

Jakarta


Jakarta EE 10
Jakarta EE Core Profile, independent individual specification development
Jakarta EE 9.1
Java SE 11 Multiple Compatible implementations available on the release date
Jakarta EE 9
New *jakarta* namespace Java SE 8 Removal of deprecated specs
Jakarta EE 8
JAKARTA EE
javax namespace, New process, New licensing New logo
Java EE 8
Eclipse Foundation infrastructure
Java EE 8
Sources, TCKs, Docs
Java
Oracle infrastructure & JCP process

SOA összetevők

- referencia architektúra
- vállalati módszertan
- irányítási modell
- fejlesztési folyamat

A szolgáltatások

- önálló entitások
- zártak
- lazán csatoltak
- felhasználóik programok
- implementációjuk nem ismert
- leíróik egy-egy szerződést definiálnak
- állapotmentesek vagy állapotőrzők

Webszolgáltatás

- Bármilyen rendszer
- Bármilyen programnyelv
- HTTP
- Mime típusok
- Web itt hálózat, nem feltétlenül Internet (Világháló, World Wide Web)
- Egységesítés, modularitás
- (verziózás)
- Kérés-válasz típusú (Request-Response, Rq/Rs)

http request methods

- Get
- Head
- Post
- Put
- Delete
- Connect
- Options
- Trace
- Patch

REST Concept

- Actually only the difference is how clients access our service. Normally, a service will use SOAP, but if you build a REST service, clients will be accessing your service with a different architectural style (calls, serialization like JSON, etc.).

- REST uses some common HTTP methods to insert/delete/update/retrieve information which is below:

- **GET** - Requests a specific representation of a resource
- **PUT** - Creates or updates a resource with the supplied representation
- **DELETE** - Deletes the specified resource
- **POST** - Submits data to be processed by the identified resourc

What is REST?

- REpresentational State Transfer
  - PhD by Roy Fielding
  - The Web is the most successful application on the Internet
  - What makes the Web so successful?
- Addressable Resources
  - Every “thing” should have an ID
  - Every “thing” should have a URI
- Constrained interface
  - Use the standard methods of the protocol
  - HTTP: GET, POST, PUT, DELETE
- Resources with multiple representations
  - Different applications need different formats
  - Different platforms need different representations (XML + JSON)
- Communicate statelessly
  - Stateless application scale

- Every “thing” has a URI

Addressability

`http://sales.com/customers/323421`
`http://sales.com/customers/32341/address`

- From a URI we know
  - The protocol (How do we communicate)
  - The host/port (Where it is on network)
  - The resource path(What resource are we communicating with

Describing a URI

http://sales.com/customers/323421
/customers/{customer-id}

- Human readable URIs: Desired but not required
- URI Parameters

```
http://sales.com/customers?zip=49009
```

- Query parameters to find other resources

```
http://sales.com/cars/mercedes/amg/e55;color=black
```

- Matrix parameters to define resource attributes

Implications of a Uniform Interface

- Intuitive
  - You know what operations the resource will support
- Predictable behavior
  - GET - readonly and idempotent. Never changes the state of the resource
  - PUT - an idempotent insert or update of a resource. Idempotent because it is repeatable without side effects.
  - DELETE - resource removal and idempotent.
  - POST - non-idempotent, “anything goes” operation
- Clients, developers, admins, operations know what to expect
  - Much easier for admins to assign security roles
  - For idempotent messages, clients don’t have to worry about duplicate messages.

Rest

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.“

RESTful Services

- Resources as URI
  - Use unique URI to reference every resource on your API
- Operations as HTTP Methods
  - GET – Queries
  - POST – Queries
  - PUT, DELETE – Inset, Update and delete
- Connectedness and Discoverability
  - Like the Web, HTTP Responses contains links to other resources

Spring

- **Spring Framework** is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- **Spring Boot** is based on the **Spring Framework**, providing auto-configuration features to your Spring applications and is designed to get you up and running as quickly as possible.
- **Spring Security** provides comprehensive security services for Java EE-based software applications. There is a particular emphasis on supporting projects built using the **Spring Framework**.

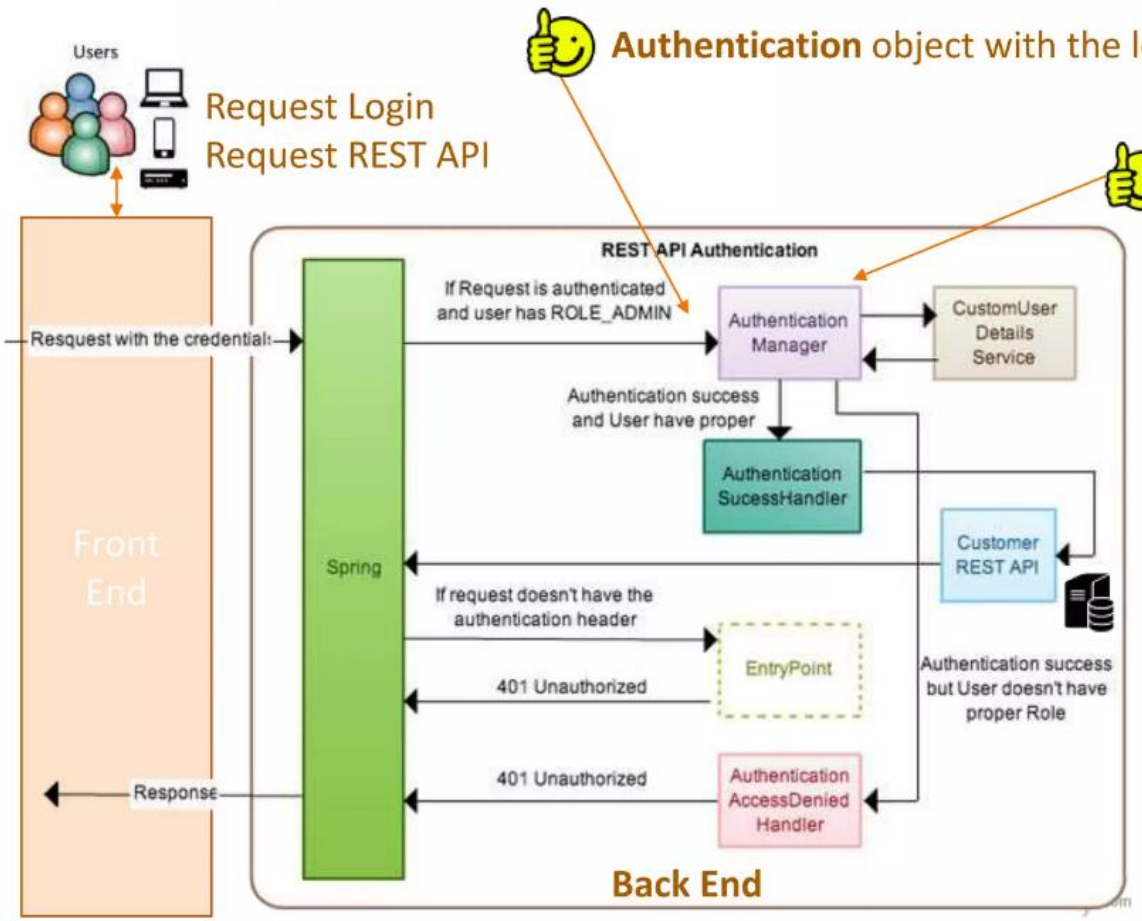
What is Spring Security

- Spring Security is a **framework** that focuses on **providing both authentication and authorization** (or "access-control") to Java web application and SOAP/RESTful web services
- Spring Security currently **supports integration** with all of the following **technologies**:
  - HTTP basic access authentication
  - LDAP system
  - OpenID identity providers
  - JAAS API
  - CAS Server
  - ESB Platform
  - .....
  - Your own authentication systems
- It is built on top of Spring Framework

Spring Security Fundamentals

- Principal
  - User that performs the action
- Authentication
  - Confirming truth of credentials
- Authorization
  - Define access policy for principal
- GrantedAuthority
  - Application permission granted to a principal
- SecurityContext
  - Hold the authentication and other security information
- SecurityContextHolder
  - Provides access to SecurityContext
- AuthenticationManager
  - Controller in the authentication process
- AuthenticationProvider
  - Interface that maps to a data store which stores your user data.
- Authentication Object
  - Object is created upon authentication, which holds the login credentials.
- UserDetails
  - Data object which contains the user credentials, but also the Roles of the user.
- UserDetailsService
  - Collects the user credentials, authorities(roles) and build an UserDetails object.

Spring Security Common Workflow

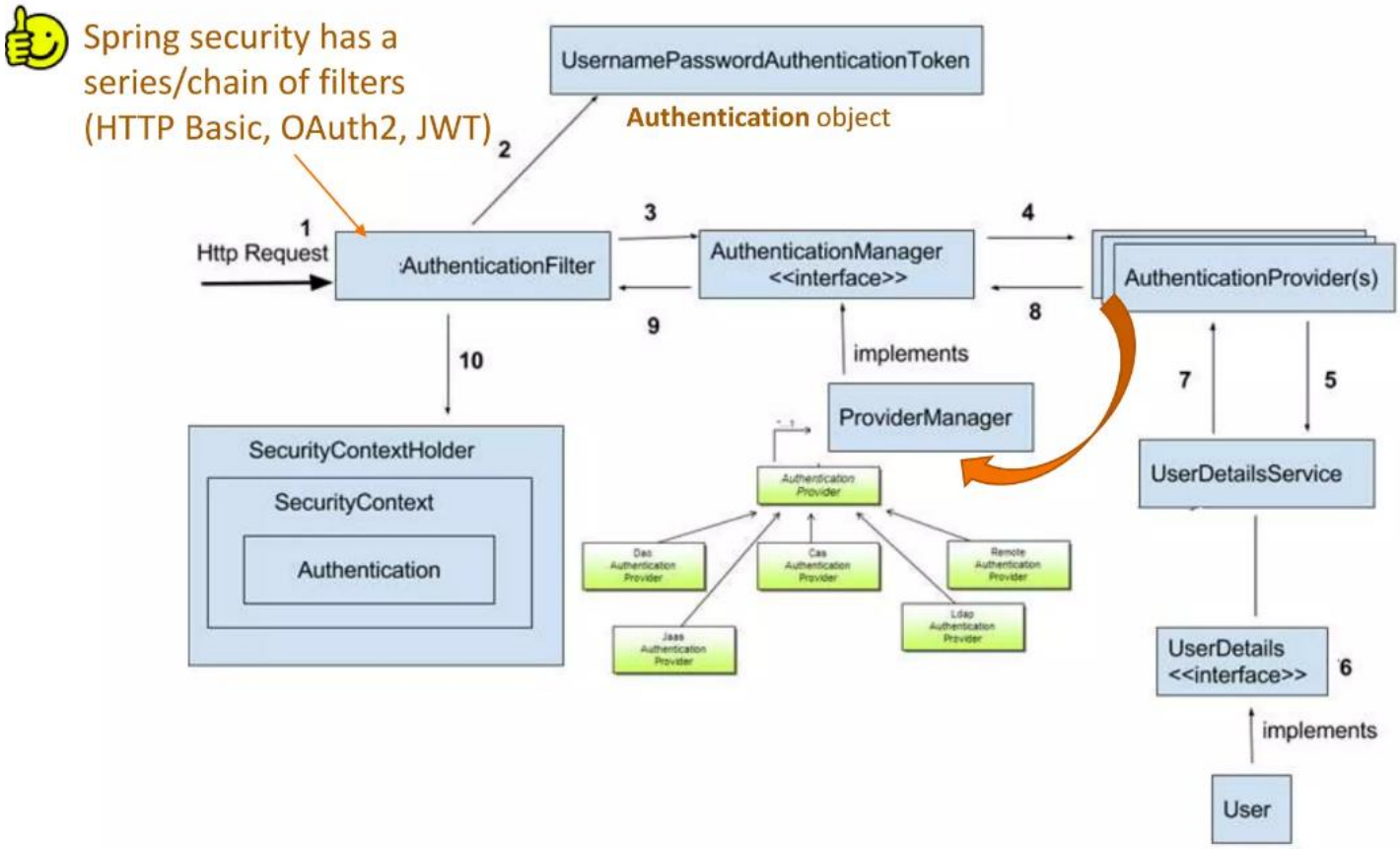


Controller in the process that it just delegates to a collection of **AuthenticationProvider** instances.

Call an object that implements the **UserDetailsService** interface, which it looks up the user data and returns a **UserDetails** object.

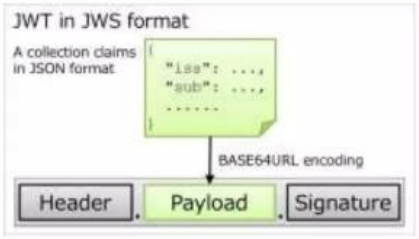
Will check that the password matches the password the user entered.

Spring Security Architecture



Spring Security Fundamentals

- **JWT (JSON Web Tokens)**
  - **It is just a token format**. JWT tokens are JSON encoded data structures containing information about issuer, subject (claims), expiration time, etc. JWT is simpler than SAML 1.1/2.0, is supported by all devices and it is more powerful than SWT (Simple Web Token). See https://jwt.io/



- **OAuth2**
  - **OAuth2 is a framework** that solves a problem when a user wants to access the data using a client software like browser-based web apps, native mobile apps or desktop apps. **OAuth2 is just for authorization**, client software can be authorized to access the resources on behalf of the end user by using an access token.

- **OpenID Connect**
  - OpenID Connect builds on **top of OAuth2** and adds authentication. OpenID Connect adds some constraints to OAuth2 like UserInfo Endpoint, ID Token, discovery and dynamic registration of OpenID Connect providers and session management. **JWT is the mandatory format for the token.**

JWT

Here are some scenarios where JSON Web Tokens are useful:

- **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

JSON Web Token Structure

In its compact form, JSON Web Tokens consist of three parts separated by dots (`.`), which are:

- Header
- Payload
- Signature

Therefore, a JWT typically looks like the following.

```
xxxxx.yyyyy.zzzzz
```

JWT Header

Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.

- **Registered claims**: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and others.

  > Notice that the claim names are only three characters long as JWT is meant to be compact.

- **Public claims**: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

- **Private claims**: These are the custom claims created to share information between parties that agree on using them and are neither *registered* or *public* claims.

An example payload could be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

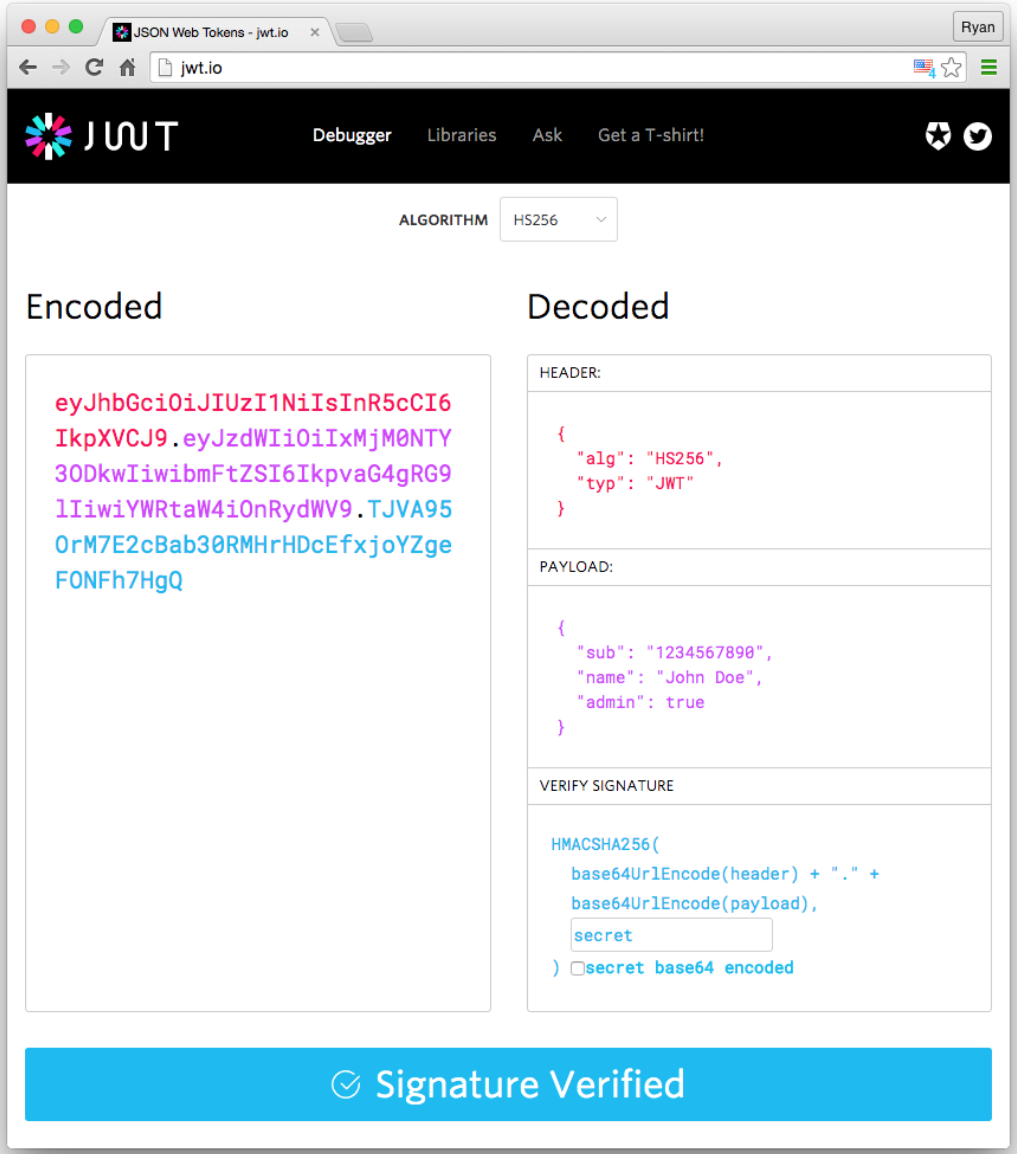
Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.


JSON Web Tokens - jwt.io
Ryan
jwt.io
JWT
Debugger
Libraries
Ask
Get a T-shirt!
ALGORITHM
HS256
Encoded
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
Decoded
HEADER:
{
"alg": "HS256",
"typ": "JWT"
}
PAYLOAD:
{
"sub": "1234567890",
"name": "John Doe",
"admin": true
}
VERIFY SIGNATURE
HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
secret
) secret base64 encoded
Signature Verified

Bearer schema

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

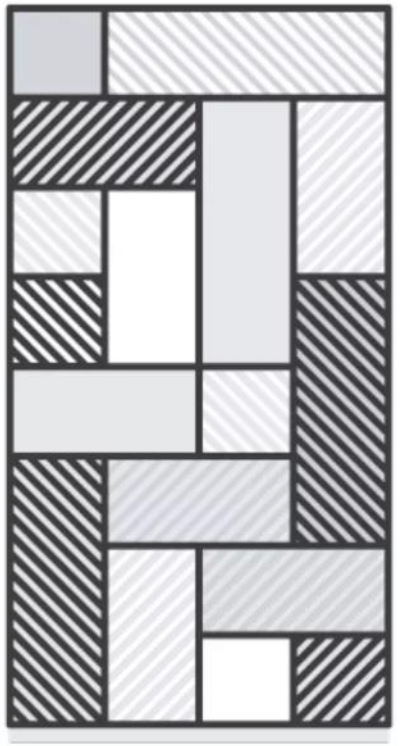
“The Monolith”

Monoliths can be **GOOD** when you are starting out

Always build to keep things as simple as possible

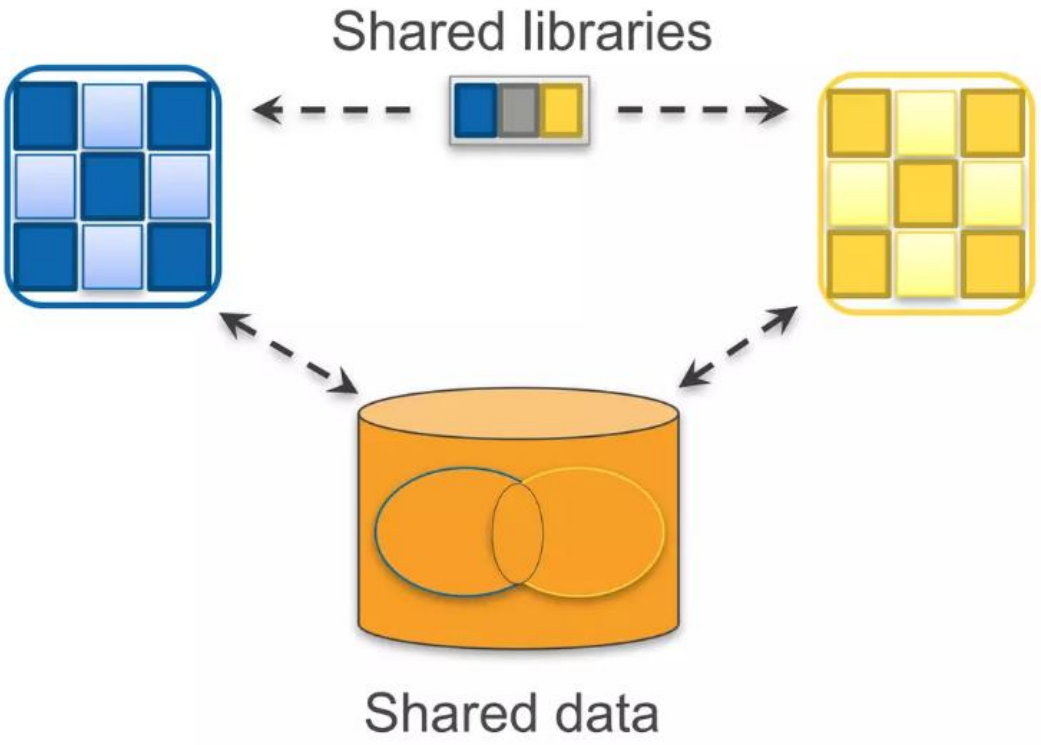
When your application or company grows, Monoliths begin slowing you down.

Let’s look at the challenges as you grow..



Challenges with monolithic software

Too much software coupling

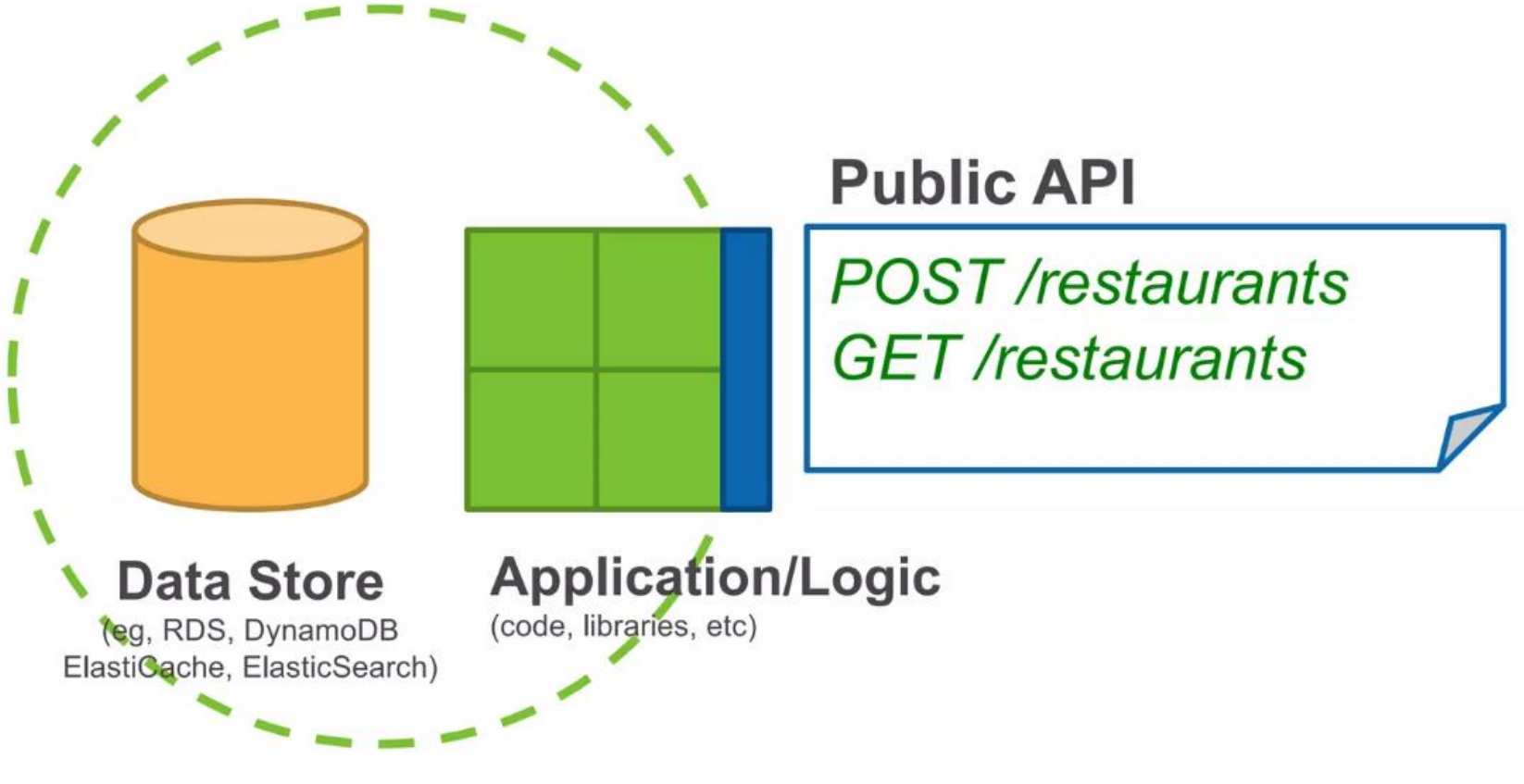


Microservice

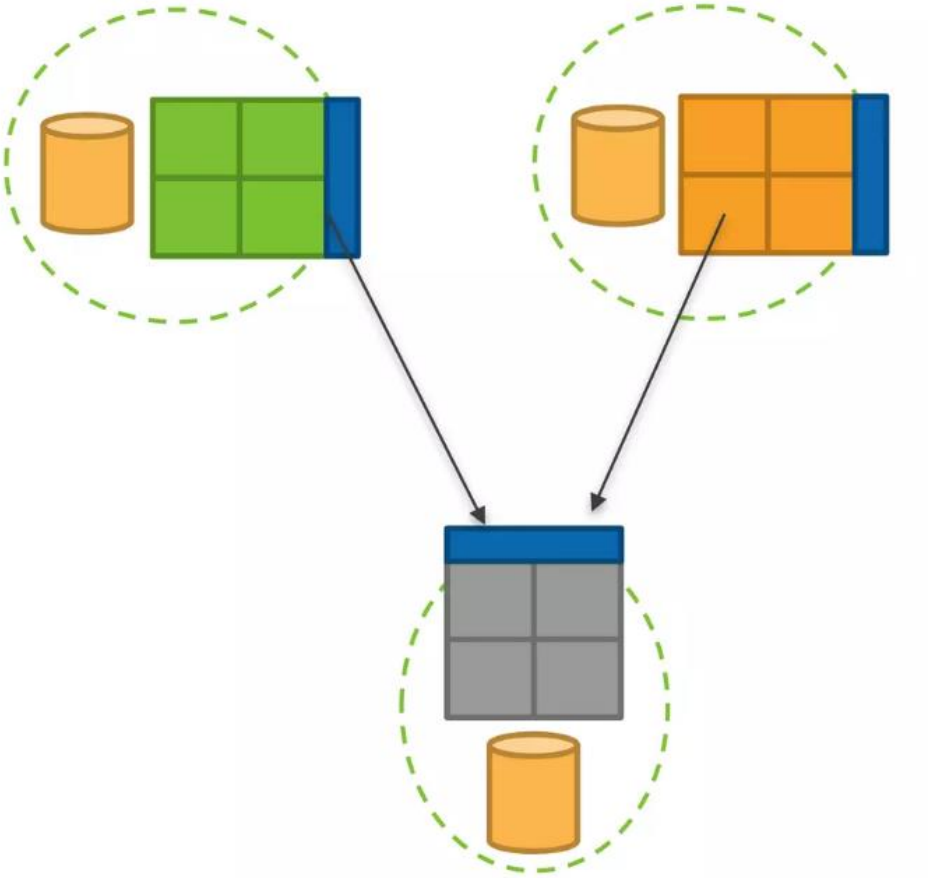
"**service-oriented architecture** composed of **loosely coupled elements** that have **bounded contexts**"

Adrian Cockcroft (VP of Cloud Architecture @ AWS, former Cloud Architect at Netflix)

Anatomy of a Microservice



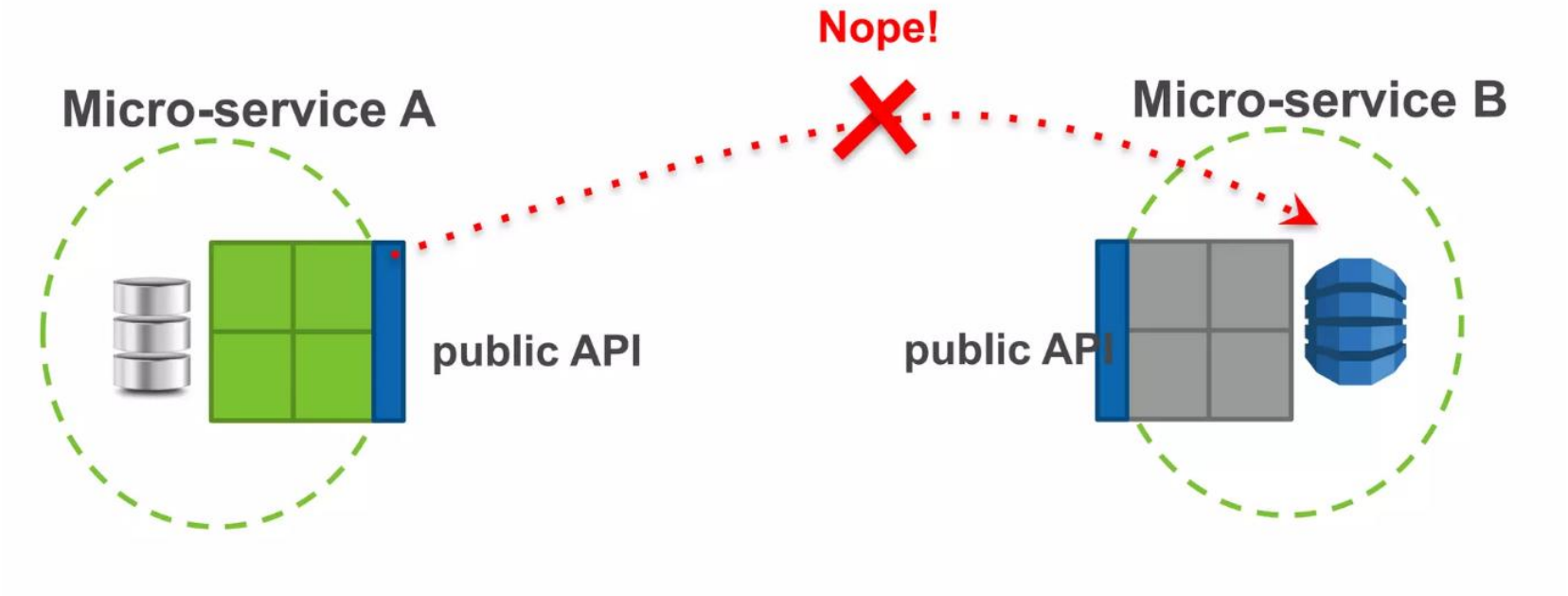
Avoid Software Coupling

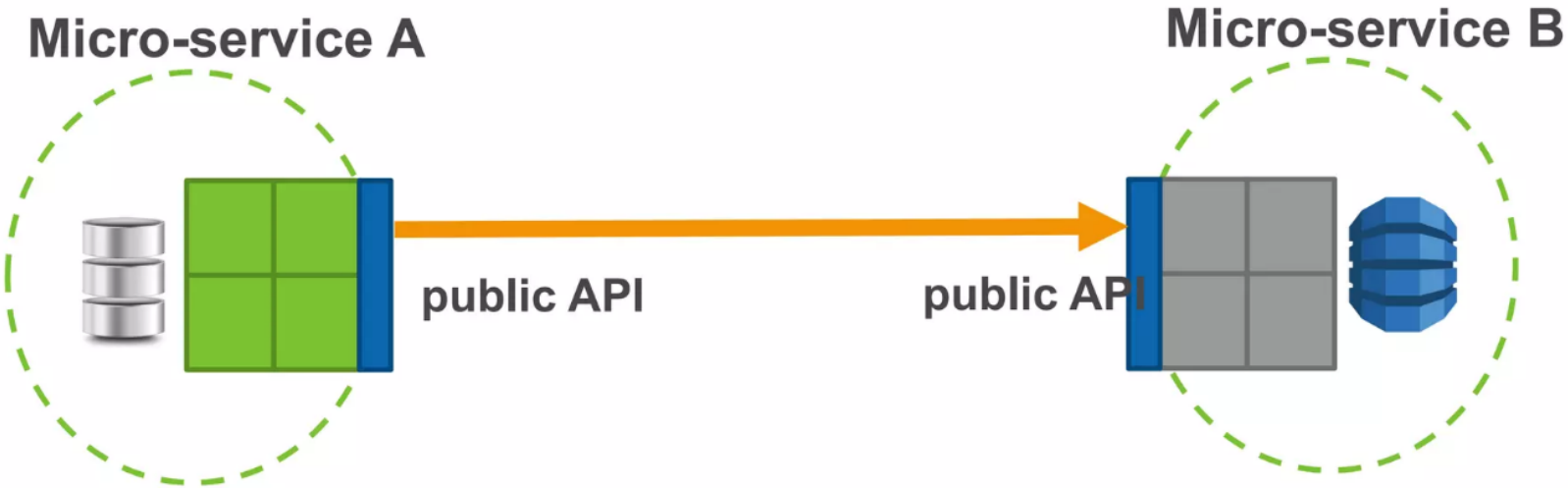


Principle 1

Microservices only rely on each other's public API

Hide your data!

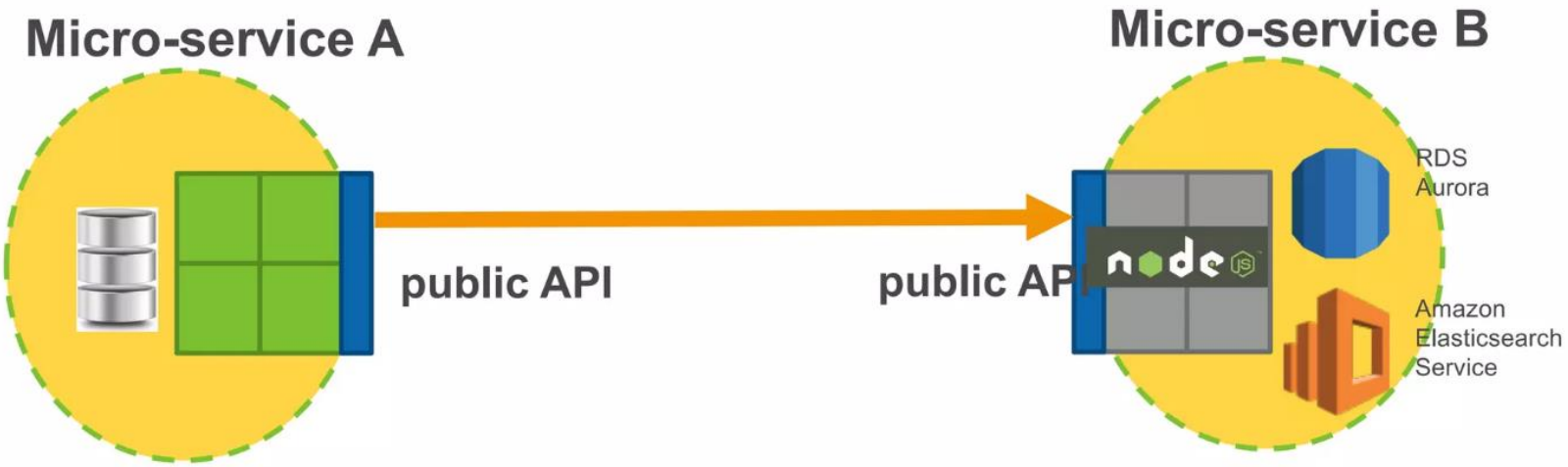



Micro-service A
Micro-service B
public API
public API

Principle 2

Use the right tool for the job

Embrace polyglot programming frameworks



Principle 2

- Embrace polyglot persistence
- Embrace polyglot programming frameworks

Principle 3: Secure your services

- **Defense-in-depth**
  - Network level (e.g. VPC, Security Groups, TLS)
  - Server/container-level
  - App-level
  - IAM policies
- **Gateway** ("Front door")
- **API Throttling**
- **Authentication & Authorization**
  - Client-to-service, as well as service-to-service
  - API Gateway: custom Lambda authorizers
  - Token-based auth (JWT tokens, OAuth 2.0)
  - IAM-based Authentication
- **Secrets management**
  - S3 bucket policies + KMS + IAM
  - Open-source tools (e.g. Vault, Keywhiz)

A Software Monolith

- One build and deployment unit
- One code base
- One technology stack (Linux, JVM, Tomcat, Libraries)

Benefits

- Simple mental model for developers
  - one unit of access for coding, building, and deploying
- Simple scaling model for operations
  - just run multiple copies behind a load balancer

Problems with monolith

- Huge and intimidating code base for developers
- Development tools get overburdened
  - refactorings take minutes
  - builds take hours
  - testing in continuous integration takes days
- Scaling is limited
  - Running a copy of the whole system is resource-intense
  - It doesn't scale with the data volume out-of-the-box
- Deployment frequency is limited
  - Re-deploying means halting the whole system
  - Re-deployments will fail and increase the perceived risk of deployment

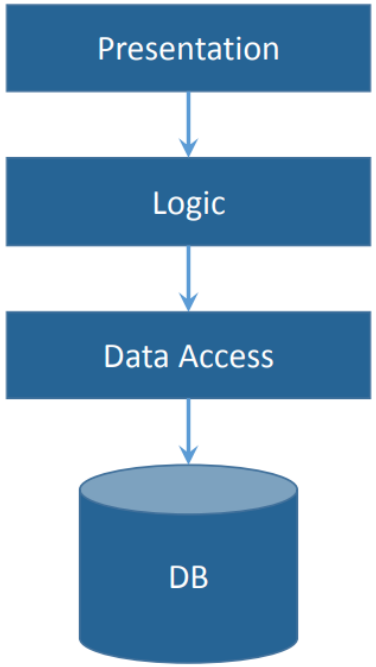
Layered Systems

A layered system decomposes a monolith into layers

- Usually: presentation, logic, data access
- At most one technology stack per layer
  - Presentation: Linux, JVM, Tomcat, Libs, EJB client, JavaScript
  - Logic: Linux, JVM, EJB container, Libs
  - Data Access: Linux, JVM, EJB JPA, EJB container, Libs

Benefits

- Simple mental model, simple dependencies
- Simple deployment and scaling model



Problems of layered systems

- Still huge codebases (one per layer)
- ... with the same impact on development, building, and deployment
- Scaling works better, but still limited
- Staff growth is limited: roughly speaking, one team per layer works well
  - Developers become specialists on their layer
  - Communication between teams is biased by layer experience (or lack thereof)

Microservice History

- 2011: First discussions using this term at a software architecture workshop near Venice
- May 2012: microservices settled as the most appropriate term
- March 2012: “Java, the Unix Way” at 33rd degree by James Lewis
- September 2012: “µService Architecture“ at Baruco by Fred George
- All along, Adrian Cockroft pioneered this style at Netflix as “fine grained SOA”

Underlying principle

On the logical level, microservice architectures are defined by a

functional system decomposition into manageable and independently deployable components

- The term “micro” refers to the sizing: a microservice must be manageable by a single development team (5-9 developers)
- Functional system decomposition means vertical slicing (in contrast to horizontal slicing through layers)
- Independent deployability implies no shared state and inter-process communication (often via HTTP REST-ish interfaces)

Independent Deployability

It enables separation and independent evolution of

- code base
- technology stacks
- scaling
- and features, too

Independent Code base

Each service has its own software repository

- Codebase is maintainable for developers – it fits into their brain
- Tools work fast – building, testing, refactoring code takes seconds
- Service startup only takes seconds
- No accidental cross-dependencies between code bases

Independent technology stacks

Each service is implemented on its own technology stacks

- The technology stack can be selected to fit the task best
- Teams can also experiment with new technologies within a single microservice
- No system-wide standardized technology stack also means
  - No struggle to get your technology introduced to the canon
  - No piggy-pack dependencies to unnecessary technologies or libraries
  - It‘s only your own dependency hell you need to struggle with ☺
- Selected technology stacks are often very lightweight
  - A microservice is often just a single process that is started via command line, and not code and configuration that is deployed to a container.

Independent Scaling

Each microservice can be scaled independently

- Identified bottlenecks can be addressed directly
- Data sharding can be applied to microservices as needed
- Parts of the system that do not represent bottlenecks can remain simple and un-scaled

Independent evolution of features

Microservices can be extended without affecting other services

- For example, you can deploy a new version of (a part of) the UI without re-deploying the whole system
- You can also go so far as to replace the service by a complete rewrite

But you have to ensure that the service interface remains stable

Standardized communication

Communication between microservices is often standardized using

- HTTP(S) – battle-tested and broadly available transport protocol
- REST – uniform interfaces on data as resources with known manipulation means
- JSON – simple data representation format

REST and JSON are convenient because they simplify interface evolution
(more on this later)

Componentiozation via Services

- Interaction mode: share-nothing, cross-process communication
- Independently deployable (with all the benefits)
- Explicit, REST-based public interface
- Sized and designed for replaceability
  - Upgrading technologies should not happen big-bang, all-or-nothing-style
- Downsides
  - Communication is more expensive than in-process
  - Interfaces need to be coarser-grained
  - Re-allocation of responsibilities between services is harder

Fallacies of Distributed Computing

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn‘t change
- There is one administrator
- Transport cost is zero
- The network is homogeneous
  - Peter Deutsch

Angular 7

- JavaScript keretrendszer, aminek segítségével SPA (Single Page Application) típusú alkalmazások készíthetők.
- Komponens alapú, melyek fa struktúrát alkotnak
- Az Angular 1.0 AngularJS elnevezést kapott, minden további verzióra egyszerűen Angular néven hivatkozunk
- "Angular is a complete rewrite of AngularJS by the same team that built AngularJS."

Single Page Application

- Az SPA egy webalkalmazás vagy weboldal, folyamatos, gyors felhasználói élményt biztosít hasonlóan, mint az asztali alkalmazások.
- Tartalmaz menüt, gombokat, blokkokat egy oldalon.
- Amikor a felhasználó bármelyikre kattint, az aktuális oldal íródik újra, így nem kell újra betölteni a további oldalakat a szerverről, ezáltal garantált a gyorsaság.

AngularJS és Angular különbségek

AngularJS	Angular
Az Angular 1.0 általános elnevezése	Az Angular 2+ általános elnevezése
JavaScript alapú nyílt forráskódú front-end web keretrendszer	TypeScript-alapú nyílt forráskódú teljeskörű web alkalmazás keretrendszer
Scope és kontroller koncepciójára épül	Komponensek hierarchiáját használja architekturális alapként
Egyszerű szintaxis, ami HTML oldalakon használható a forrás helyének megjelölésével	Különböző kifejezés szintaxisokat használ, például „[]”-t tulajdonságkötésre, „()”-t eseménykötésre
Egyszerű JavaScript fájl, ami HTML oldalakon használható, viszont nem támogatja a szerver oldali programozási nyelvet	A Microsoft TypeScript nyelvét használja, ami Osztály alapú Objektum Orientált Programozást, Statikus Típusokat, Generikusokat támogat, ami a szerver oldali programozási nyelveknél is megtalálható
Nem támogatja az oldalak dinamikus betöltését	Támogatja az oldalak dinamikus betöltését

Funkciói – támogatott platformok

- Asztali alkalmazások : használható hagyományos Windows, Mac, Linux rendszerekre készült alkalmazásokhoz
- Natív alkalmazások: használható natív alkalmazások készítéséhez Cordova, Ionic, NativeScript segítségével
- Progresszív web alkalmazások: a leggyakoribb alkalmazástípus, amit az Angular keretrendszerrel készítenek. Támogatja a modern web platform lehetőségeit, nagy teljesítményű, offline, telepítési nélküli alkalmazásokat.

Angular verziók

- AngularJS: 2010
- Angular2: 2014-ben mutatták be, mint az Angular teljes újraírása számos változtatással, így a fejlesztők nem rajongtak érte. Az első kész verziót 2016 szeptember 14-én adták ki.
- Angular4: 2016 decemberében jelentették be, 2017 Márciusában jelent meg. Új HTTP kezelési rendszert kapott.
- Angular5: 2017 novemberben jelent meg, PWA, valamint Material Design fejlesztésekkel
- Angular6: 2018 májusban jelent meg, ng update, ng add, Angular Elements, Angular Material
- Angular7: 2018 októberben jelent meg, teljesítménynövekedés, Angular Material, Virtual Scrolling

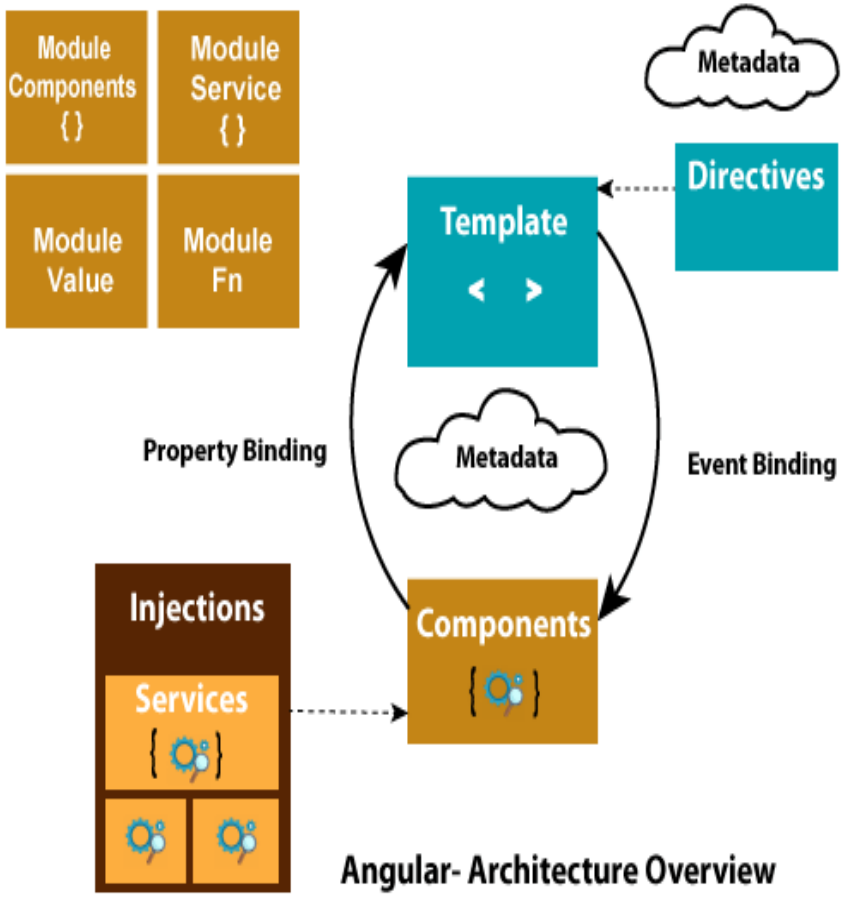
Fájlok

- Src mappa: A forrásfájlokat tartalmazza, amiket az angular alkalmazás használ
- App mappa: Az alkalmazás komponenseihez tartozó fájlok
- App.component.css: css kódokat tartalmaz a komponenshez
- App.component.html: html kódot tartalmaz a komponenshez. Ez a template állomány, amit az angular adatkötéshez használ
- App.component.spec.ts: egységtesztelési állomány a komponenshez. További egységtesztekkel együtt használatos. Az Angular CLI-ből ng test paranccsal futtatható.
- App.component.ts: A megjelenítési logikát tartalmazza a komponenshez
- App.module.ts: a weboldalhoz tartozó függőségeket tartalmazza. Itt definiálhatók a szükséges modulok.

Angular CLI parancsok

Command	Alias	Description
add		Külső könyvtár hozzáadása
build	b	Lefordítja az alkalmazást a kimeneti könyvtárba
config		Az angular.json állomány értékeit módosítja vagy lekérdezi
doc	d	Megnyitja az Angular dokumentációját
e2e	e	Buildeli és elindítja az alkalmazást, valamint elindítja a teszteket a Protractor segítségével
generate	g	Generál vagy módosít állományokat a séma alapján
help		Elérhető parancsok listája
lint	l	Lint eszközök futtatása az adott könyvtárban
new	n	Új Angular alkalmazást hoz létre
run		Arcitect célt futtat opcionális egyedi konfigurációval
serve	s	Buildeli és elindítja az alkalmazást, fájlváltozás esetén is

Angular 7 architektúra



Komponensek

- A komponensek és szolgáltatások egyszerű osztályok dekorátorokkal, amik megjelölik a típusaikat és metaadatot szolgáltatnak amit az Angular használ fel.
- Minden Angular alkalmazás legalább egy komponensből áll, aminek a neve root komponens és oldalhierarchia és dom tartozik hozzá. Minden komponens definiál egy osztályt, ami alkalmazásadatot és logikát tartalmaz, és kapcsolódik egy HTML templatehez, ami egy nézetet ad meg a célkörnyezet számára.

A komponens osztály metaadatai

- A komponens osztályhoz tartozó metaadat kapcsolódik a templatehez, ami a nézetet adja meg. A template kombinálja a hagyományos HTML-t az Angular direktívákkal és kötési jelölésekkel, amik által az Angular módosíthatja a HTML-t még renderelés előtt.
- A szolgáltatás osztály metaadata információt szolgáltat az Angularnak, aminek segítségével elérhetővé válhat a komponensben a dependency injection által.

Modulok

- Az Angular 7 NgModulok különböznek a többi JavaScript modultól. Minden Angular 7 alkalmazás tartalmaz egy gyökér modult, amit AppModule-nak nevezünk. Ebben kap helyet a bootstrap mechanizmus, ami az alkalmazást indítja
- Minden Angular 7 alkalmazás több funkcionális modulból áll.
- Az NgModule funkciókat importál további NgModulokból, pont úgy, mint a hagyományos JavaScript modulok
- Az NgModul megengedi, hogy funkciói kinyerhetők legyenek más NgModulok által

Template, Direktívák, Adatkötés

- Az Angular 7-ben templateket használhatunk, hogy kombináljuk a HTML-t az Angular jelölésekkel és módosítsuk a HTML elemeket megjelenítés előtt. A template direktívák program logikát szolgáltatnak és a jelöléseket kötik az alkalmazás adatokhoz és a DOMhoz.
- Adat kötések
  - Esemény kötés: Az esemény kötéseket akkor használjuk, ha az alkalmazás reagál a felhasználói inputra, frissítve az alkalmazás adatokat.
  - Tulajdonság kötés: A tulajdonság kötés arra használható, hogy adatot továbbítsunk a komponens osztályból.

Szolgáltatások és Dependency Injection

- Angular 7-ben a fejlesztők szolgáltatás osztályokat hoznak létre adatnak vagy logikának, amit nem egy nézethez szeretnének kapcsolni, hanem meg akarják osztani a komponensek között.
- A Dependency Injection által a komponens osztályok kis méretűek és hatékonyak lehetnek. A DI nem tölt be adatokat a szerverről, nem validálja az inputot, nem logol a konzolra, csak a feladatokat adja a szolgáltatásnak.

Routing

- Az Angular 7 Routere egy NgModul, ami egy szolgáltatáson keresztül biztosítja a fejlesztőknek az alkalmazáson belüli státuszok és nézet hierarchiák közötti navigálást.
- Ugyanúgy működik, mint a böngészőbeli navigálás:
  - Megadható URL a címrészben, és a böngésző navigálni fog az adott oldalra
  - Rákattinthatunk linkra az oldalon, amivel navigálhatunk az új oldalra
  - A böngésző előre és vissza gombjaival navigálunk a megfelelő oldalra

Direktívák

- Komponens direktívák: A fő osztályban használatosak, arról szolgáltatnak információt, hogy hogyan kell feldolgozni, létrehozni és használni a komponenst futásidőben.
- Strukturális direktívák: * karakterrel kezdődnek, DOM elemek manipulálására és megváltoztatására használható. Pl.: *ngIf, *ngFor.
- Attribútum direktívák: Segítségükkel megváltoztatható a DOM elemek kinézete és viselkedése. Pl.: ngClass, ngStyle.

Attribútum és strukturális direktívák különbségei

- Az attribútum direktívák úgy néznek ki, mint a hagyományos HTML attribútumok, és főképp adat- és eseménykötésre használjuk őket.
- A strukturális direktívák * karakterrel kezdődnek, és másképpen néznek ki.
- Az attribútum direktívák csak arra az elemre vannak hatással, amelyhez hozzá lettek adva.
- A strukturális direktívák a DOM teljes területére kihatnak.

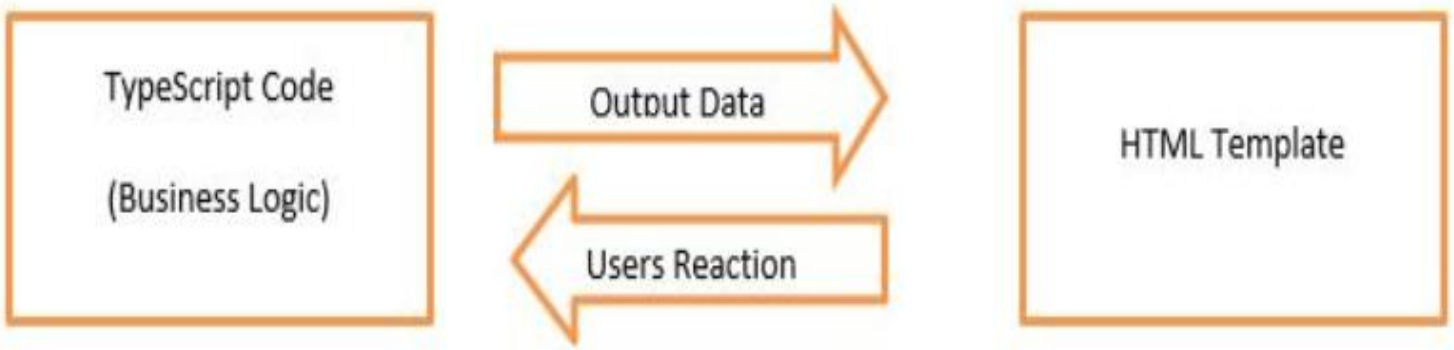
Angular adatkötés

- Az adatkötés jelentős funkciója az Angularnak, mivel kommunikációhoz használható a TypeScript kód és a felhasználói komponensek között. Pl.: HTML felület

Adatkötés típusai

- Egy utas adatkötés: a HTML template változik, amikor a TypeScript kód változik. A model értékei megjelennek a view rétegen, viszont nem frissíthető a model a view-ból. Az Angular interpoláció, string interpoláció, tulajdonságkötés, eseménykötés mind egy utas kötés.
- Két utas adatkötés: Definiált egy automatikus szinkronizáció a model és a view réteg között, így a változás mindkét komponensre kihat. Amikor a model változik, kihat majd a view-ra, és fordítva is. Ez egyből megtörténik, így a HTML template és a TypeScript kód is folyamatosan frissül.

Két utas adatkötés



String interpoláció

- A string interpoláció dinamikus adatmegjelenítést tesz lehetővé HTML templaten. Ahányszor a component.ts állomány változik, a változtatások kikerülnek a HTML oldalra.

String interpoláció vs Tulajdonság kötés

- A string interpoláció és a tulajdonság kötés is egy utas adatkötés, így egy irányban áramlanak az adatok a componenstől a HTML felé.
- A string interpoláció egy speciális szintaxis, amit tulajdonság kötéssé alakít át az Angular.
- Ha stringeket szeretnénk konkatenálni, interpolációt használhatunk tulajdonság kötés helyett.

Esemény kötés

- Az Angularban lehetőség van függvényeket eseményekhez kötni. Ezt a () karakterekkel érhetjük el.

Beépített pipe-ok

- Lowercase
- Uppercase
- Date
- Currency
- Json
- Percent
- Decimal
- Slice

Angular 7 formok

- Az angular formok a felhasználói inputok kezelésére használható. Használható például bejelentkezéshez, profil frissítéshez, adatbevitelhez.
- Kétféle megközelítés: reaktív formok és sablon vezérelt formok

Reaktív formok

- Robusztus formok
- Skálázhatók, újrafelhasználhatók és tesztelhetők
- Akkor használatos, ha a formok az alkalmazás kulcsfontosságú részeit képezik, vagy ha reaktív mintákkal készült a program.

Sablonvezérelt formok

- A sablonvezérelt formok akkor használhatók a legjobban, ha csak egyszerű formok léteznek az alkalmazásban. Pl.: e-mail lista feliratkozás.
- Nagyon egyszerűen használhatók, viszont nem annyira skálázható.
- Csak akkor használatos, ha végtelenül egyszerű és logikus formjaink vannak, amik sablonok alapján készülhetnek.

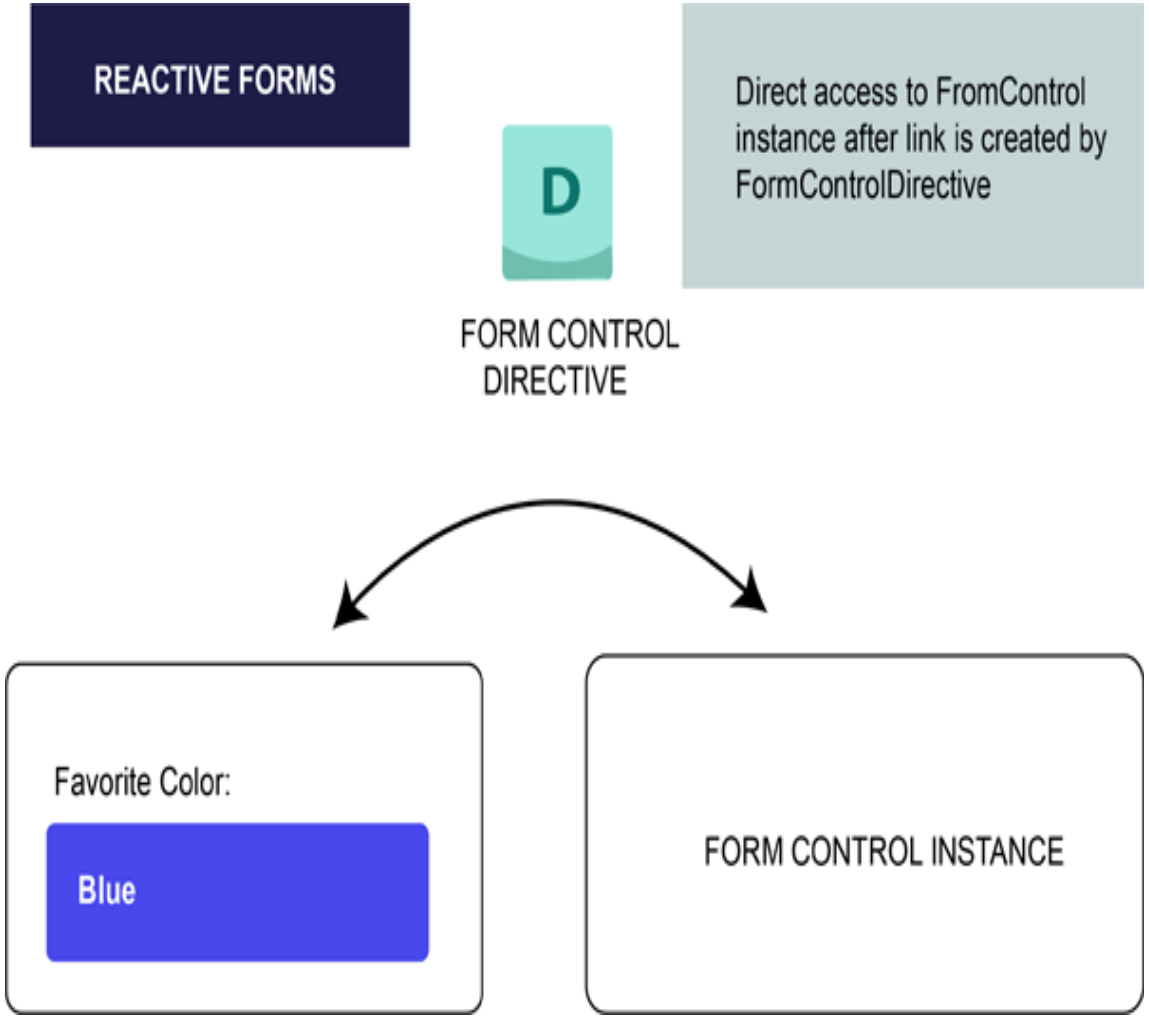
Reaktív és sablonvezérelt form különbségek

Összehasonlítási tényező	Reaktív form	Sablonvezérelt form
Felépítés	A reaktív formok explicit módon definiáltak. A komponens osztályban hozzuk létre őket.	Direktívák segítségével hozzuk létre őket.
Adatmodell	Strukturált	Strukturálatlan
Kiszámíthatóság	Szinkron	Aszinkron
Validáció	Függvénnyel	Direktívákkal
Változékonyság	Állandó	Változtatható
Skálázhatóság	Alacsony szintű API	Absztrakció az APIn felül

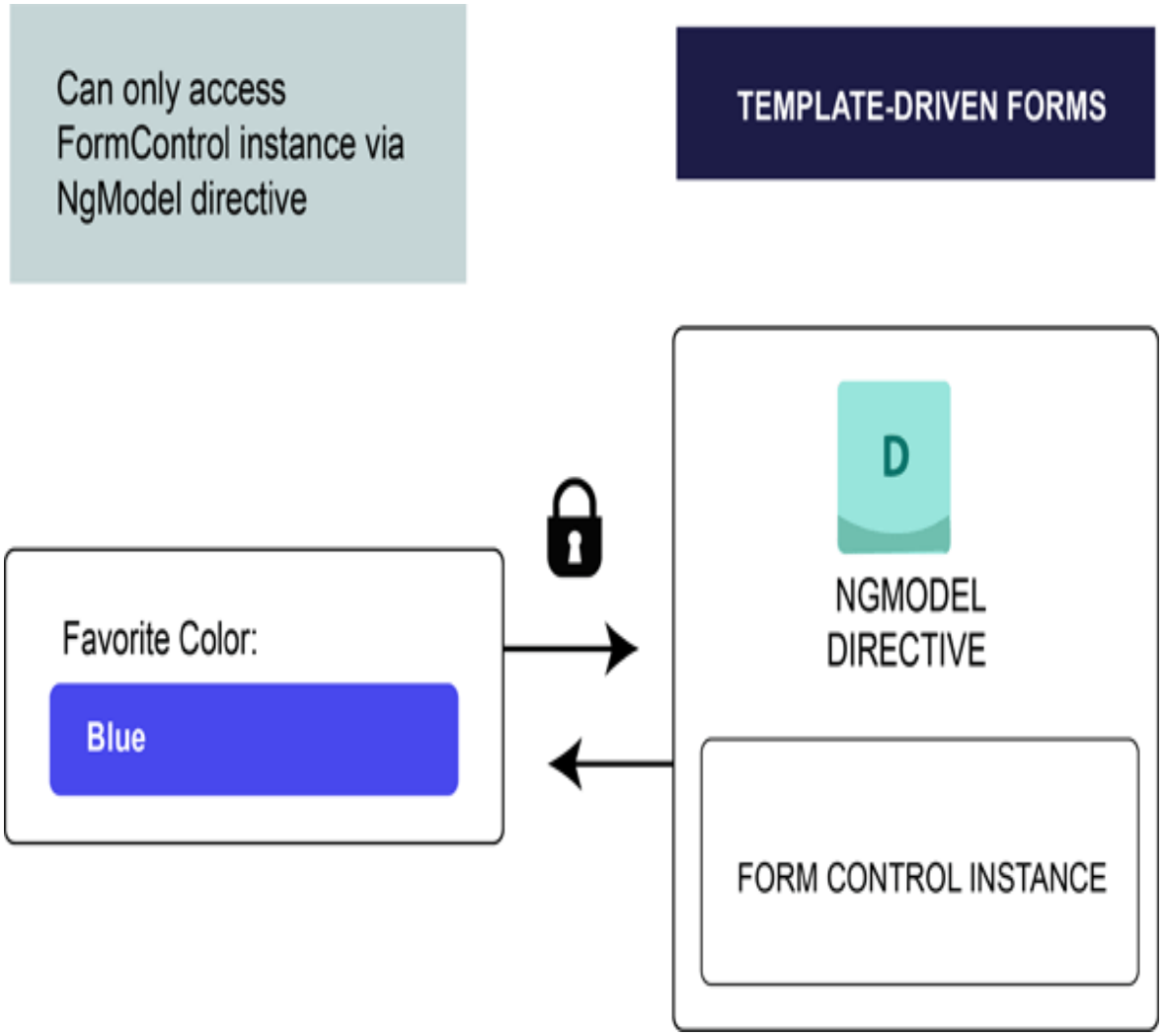
Hasonlóságok a reaktív és a sablonvezérelt formok között

- FormControl: Arra használjuk, hogy egy form control elem értékeit és a validáció státuszát figyeljük
- FormGroup: Arra használjuk, hogy egy form control kollekció értékeit és a státuszát figyeljük
- FormArray: Arra használjuk, hogy egy form control tömbben az értékeket és a státuszokat figyeljük
- ControlValueAcessor: Áthidalást jelent az Angular FormControl példányok és a natív DOM elemek között

Reaktív formok felépítése - model



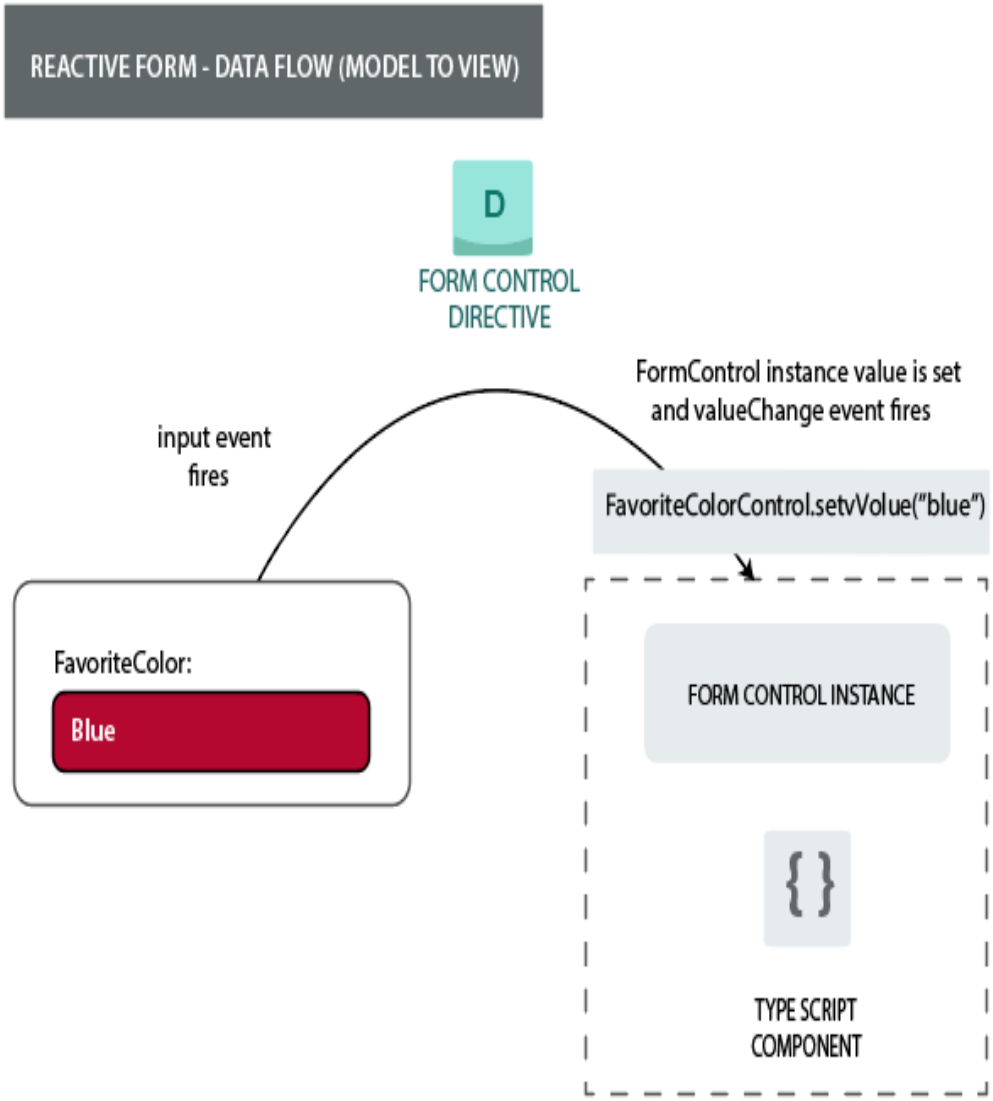
Sablonvezérelt formok felépítése - model



Adatfolyam reaktív formoknál

- A felhasználó kitölti az input mezőket
- Elindul egy „input” esemény a legutóbbi beírt értékkel
- Az értéket azonnal megkapja a FormControl példány
- A FormControl új értéket adhat a valueChanges megfigyelővel
- Mindenki, aki feliratkozott a valueChangesre, megkapja az új értéket

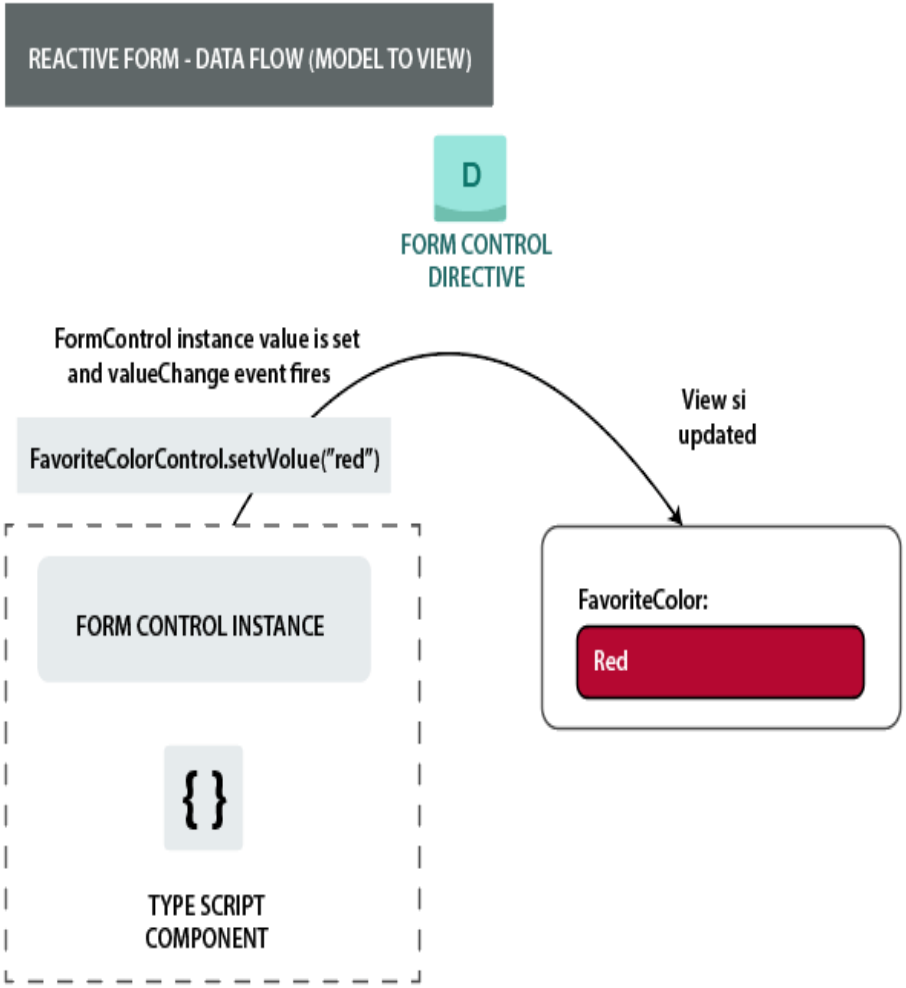
Adatfolyam reaktív formoknál



Adatfolyam reaktív formoknál – model-től view-ig

- A felhasználó meghívja a kívánt függvényt, ami frissíti a FormControl értéket.
- A FormControl példány új értéket ad a valueChanges megfigyelőnek.
- A valueChanges megfigyelő feliratkozói megkapják az új értéket.
- Végül a form beviteli mezője frissül az új értékekkel.

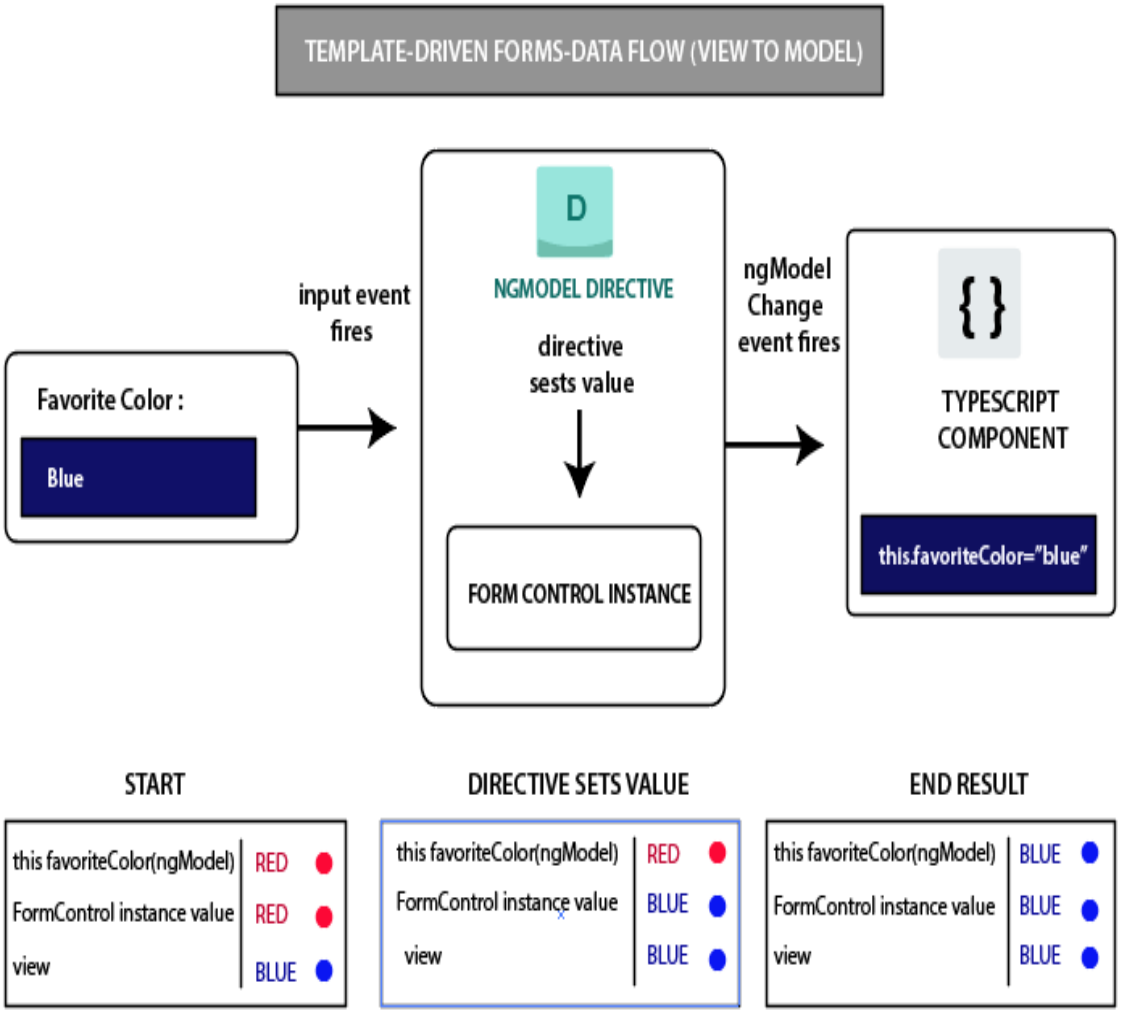
Adatfolyam reaktív formoknál – model-től view-ig



Adatfolyam sablonvezérelt formoknál

- A felhasználó kitölti az input mezőt
- Input esemény jön létre, ami tartalmazza a bevitt értékeket
- A FormControl példány setValue() függvényei meghívódnak
- A FormControl átadja az új értéket a valueChanges megfigyelőnek
- A valueChanges feliratkozói megkapják az új értéket
- Meghívódik az NgModel.viewToModelUpdate() függvény, ami ngModelChange eseményt hoz létre
- A komponens sablon két utas adatkötést használ az inputmező értékére, az új értéket az ngModelChange eseményből véve

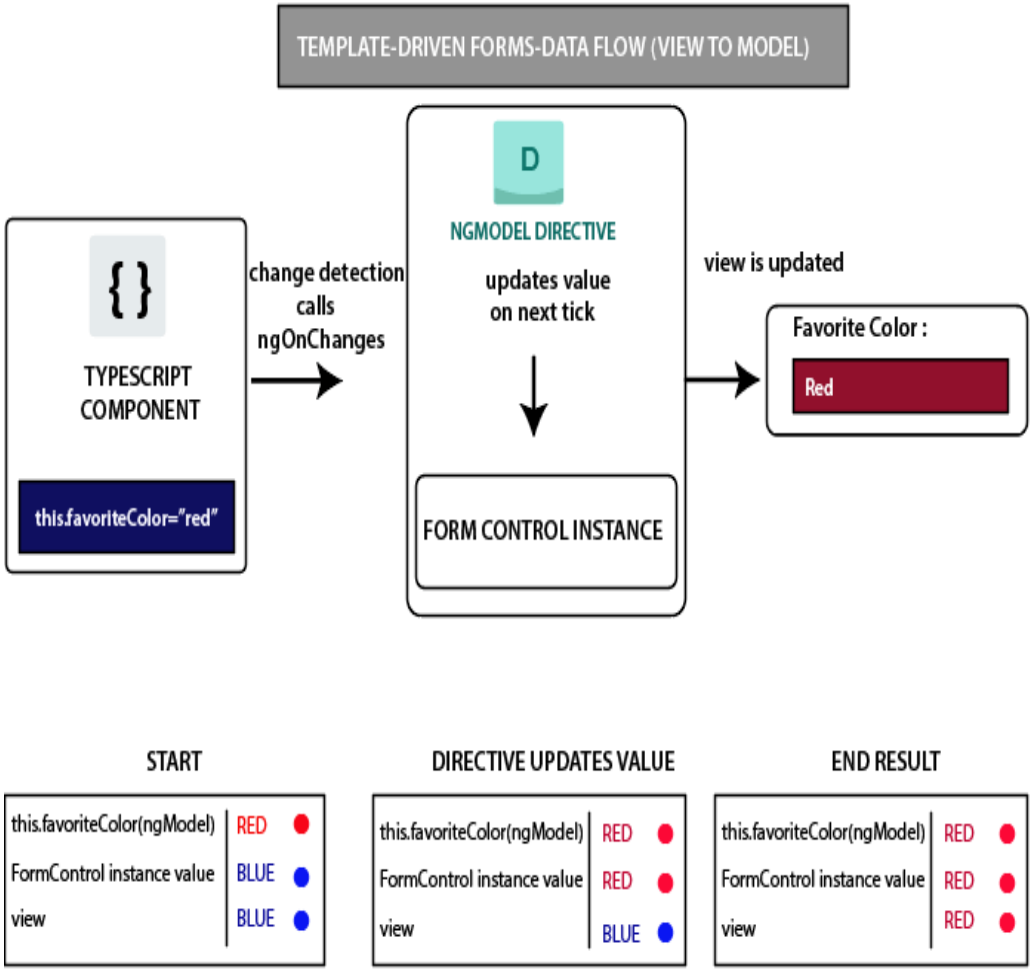
Adatfolyam sablonvezérelt formoknál



Adatfolyam sablonvezérelt formoknál – model-től view-ig

- A beviteli mező értékeit megkapja a komponens, felülírva a sajátjait
- A változás felismerése elkezdődik
- Ezalatt az ngOnChanges életciklus elindul az NgModel direktíva példányon, mivel az egyik input értéke változott
- Az ngOnChanges függvény elindít egy aszinkron taskot a FormControl példány értékének felülírására
- A változás felismerése befejeződött
- A FormControl példány beállítása befejeződik
- A FormControl példány új értéket ad a valueChanges megfigyelőnek.
- A valueChanges megfigyelő feliratkozói megkapják az új értéket.
- Végül a form beviteli mezője frissül az új értékekkel.

Adatfolyam sablonvezérelt formoknál – model-től view-ig



Angular vs React

Comparison Index	Angular	React
History	Angular is a TypeScript based JavaScript framework. It is written in TypeScript. **Angular is developed and maintained by Google** and known as a "*Superheroic JavaScript MVWFramework*". Angular is a complete rewrite of AngularJS. AngularJS was released in 2010 and it takes almost 6 years to release its second version Angular 2 (a complete rewrite). The latest version of Angular is Angular 8 now. Google AdWords which is one of the important project of Google uses Angular, so it is going to be big in upcoming years.	React is not a framework. It is a JavaScript library developed and maintained by Facebook and described as "a JavaScript library for building user interfaces. React was released in 2013 and after that it is being used at Facebook.
Architecture	Angular is a full MVC (Model, View, and Controller) framework. Angular is considered a framework because it provides strong facilities like how to structure your application. In Angular, you don't need to decide routing libraries. Most prominent features of Angular: ◦ Provides templates, based on an extended version of HTML. ◦ Provides XSS protection. ◦ Provides Dependency injection. ◦ Provides Ajax requests by @angular/HTTP. ◦ @angular/router for Routing. ◦ Component CSS encapsulation. ◦ Utilities for unit-testing components. ◦ @angular/forms for building forms.	React is a simple JavaScript library (just the View) but it gives you much more freedom. React facilitates you to choose your own libraries. Most prominent features of React: ◦ React uses JSX, an XML-like language built on top of JavaScript instead of classic templates. ◦ XSS protection. ◦ No dependency injection. ◦ Fetch for Ajax requests. ◦ Utilities for unit-testing components. React also provides some popular libraries to add functionalities: ◦ React-router for routing. ◦ Redux or MobX for state management. ◦ Enzyme for additional testing utilities.

Used DOM	Angular uses **regular DOM**. The regular DOM updates the entire tree structure of HTML tags. It doesn't make difference in a simple real app but if you are dealing with large amount of data requests on the same page (and the HTML block is replaced for every page request), it affects the performance as well as the user's experience.	React uses **virtual DOM** which makes it **amazing fast**. It was said the most prominent feature of React when it was released. It updates only the specific part within a block of HTML codes. The virtual DOM looks only at the differences between the previous and current HTML and changes only that part which is required to be updated.
Used Templates	Angular uses enhanced HTML templates with Angular directives i.e. "ng-if" or "ng-for" etc. It is quite difficult because you have to learn its specific syntax.	React uses UI templates and inline JavaScript logic together which was not done by any company before. This is called JSX. React uses component which contains both the markup AND logic in the same file. React also uses an XML-like language which facilitates developers to write markup directly in their JavaScript code. JSX is a big advantage for development, because you have everything in one place, and code completion and compile-time checks work better.
Data Binding	Angular provides two-way data binding. When you change the model state, then the UI element changes. In Angular, if you change the UI element, then the corresponding model state changes as well. Additionally, if you change the model state, then the UI element changes.	React provides one-way data binding. In React first the model state is updated, and then it renders the change in the UI element. When you change the UI element, the model state does not change.
TypeScript vs JavaScript	Angular is written in TypeScript, so you must be comfortable with TypeScript before using Angular.	React uses JavaScript which is a dynamically-typed language, so you don't have to define the variable's type. It makes it easy to use.
Scalability	Angular is easy to scale.	React is more scalable than Angular.
Speed	Angular is fast as compared to old technologies but React is faster than Angular.	React is faster than Angular.

Size	The size of Angular is large, so it takes longer load time and performance on mobile.	The size of React is smaller than Angular, so it is a little bit faster.
Company Using	◦ Google ◦ Nike ◦ Forbes ◦ Upwork ◦ General Motors ◦ HBO ◦ Sony etc.	◦ Facebook ◦ Airbnb ◦ Uber ◦ Netflix ◦ Instagram ◦ WhatsApp ◦ Dropbox etc.