rAppid:js -

the declarative RIA Javascript MVC Framework

Tony Findeisen & Marcus Krejpowicz

*


Marcus Krejpowicz
github.com/krebbl
@krebbl
Tony Findeisen
github.com/it-ony
@tonyfindeisen

Schedule

- **Tuesday** 17:00 - 19:00

  basics, concepts, principles, features, installation

- **Wednesday** 08:00 - 10:00

  extended features necessary for RIA

- **Thursday** 08:00 - 17:00

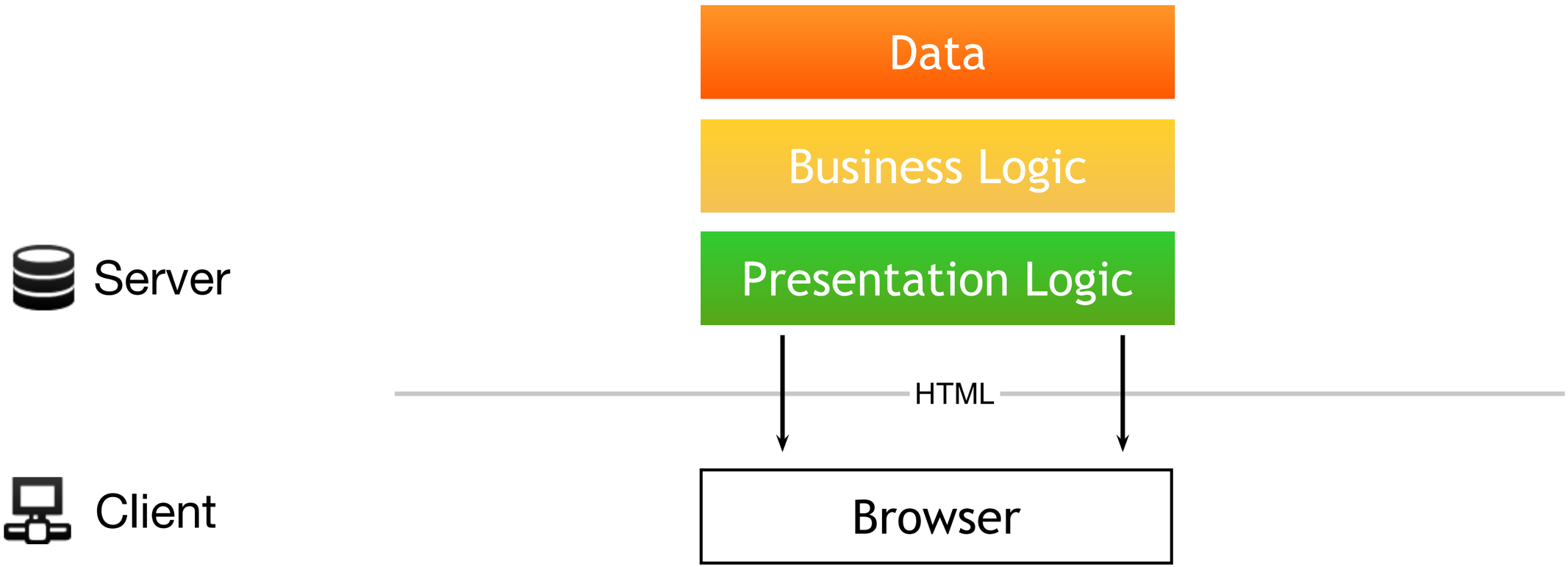
  Reimplementation of the “Wizard” as Single Page Application

Schedule **Tuesday**

1. Why single page applications?
2. Principles, Features
3. Installation, Project structure
4. XAML, Components, Shadow Dom
5. Defining classes
6. Bindings
7. Model & Entity, Schema, Validation

Why single page applications?

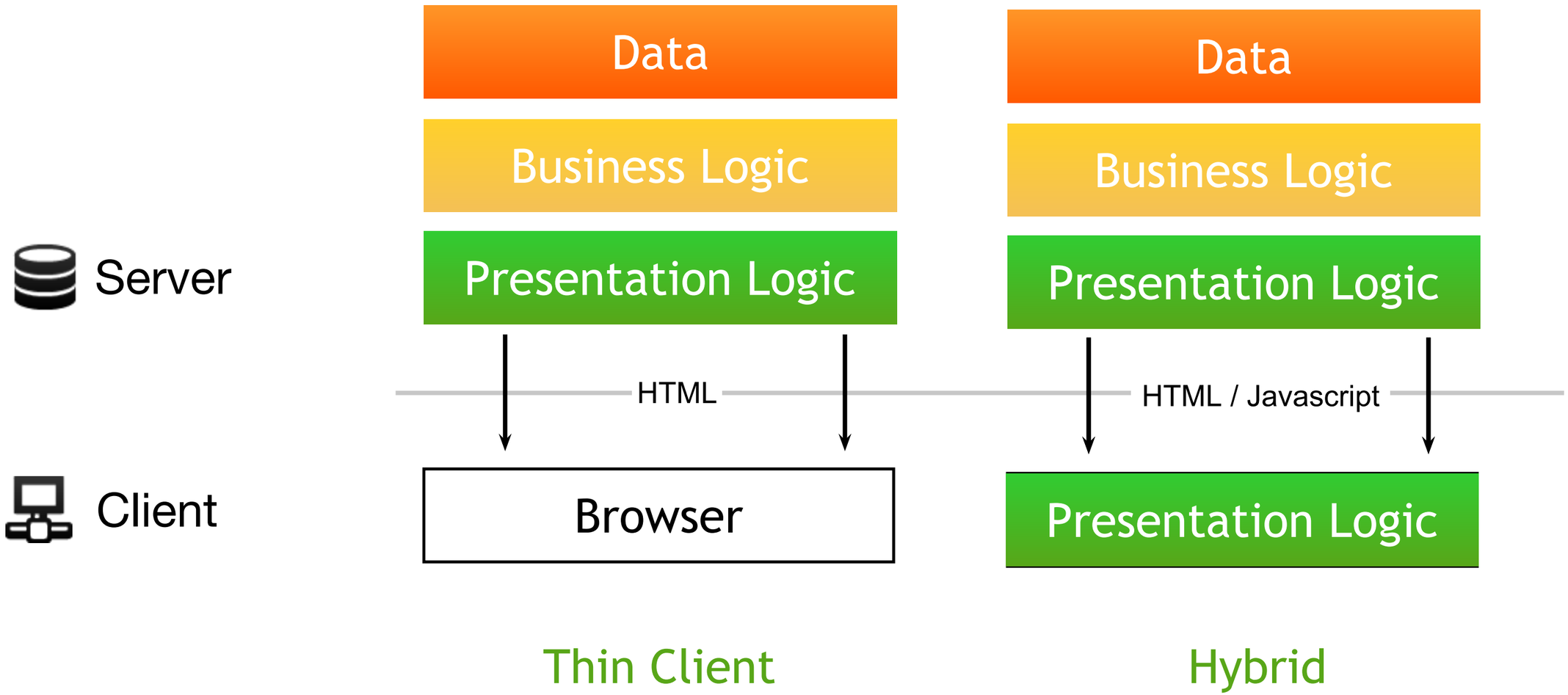
Web Applications



Thin Client

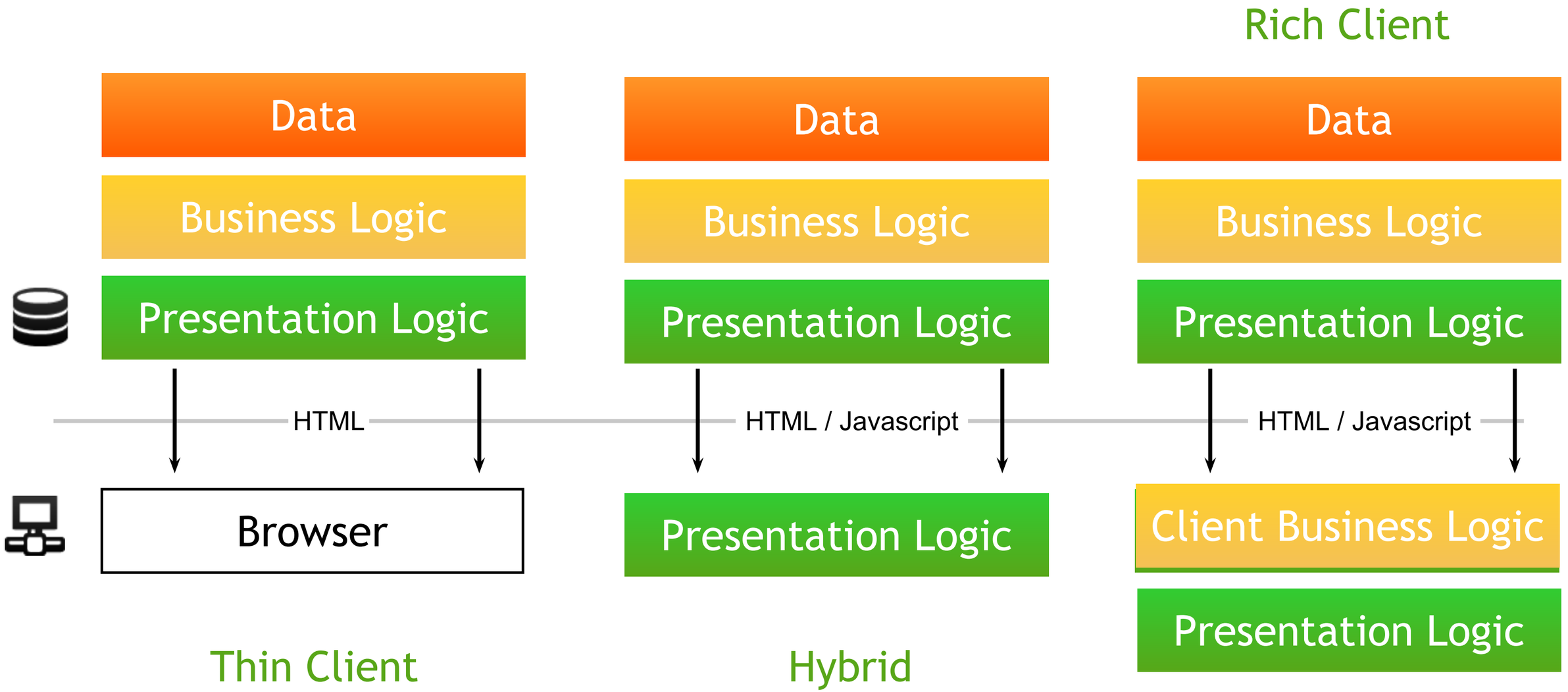
*

Web Applications



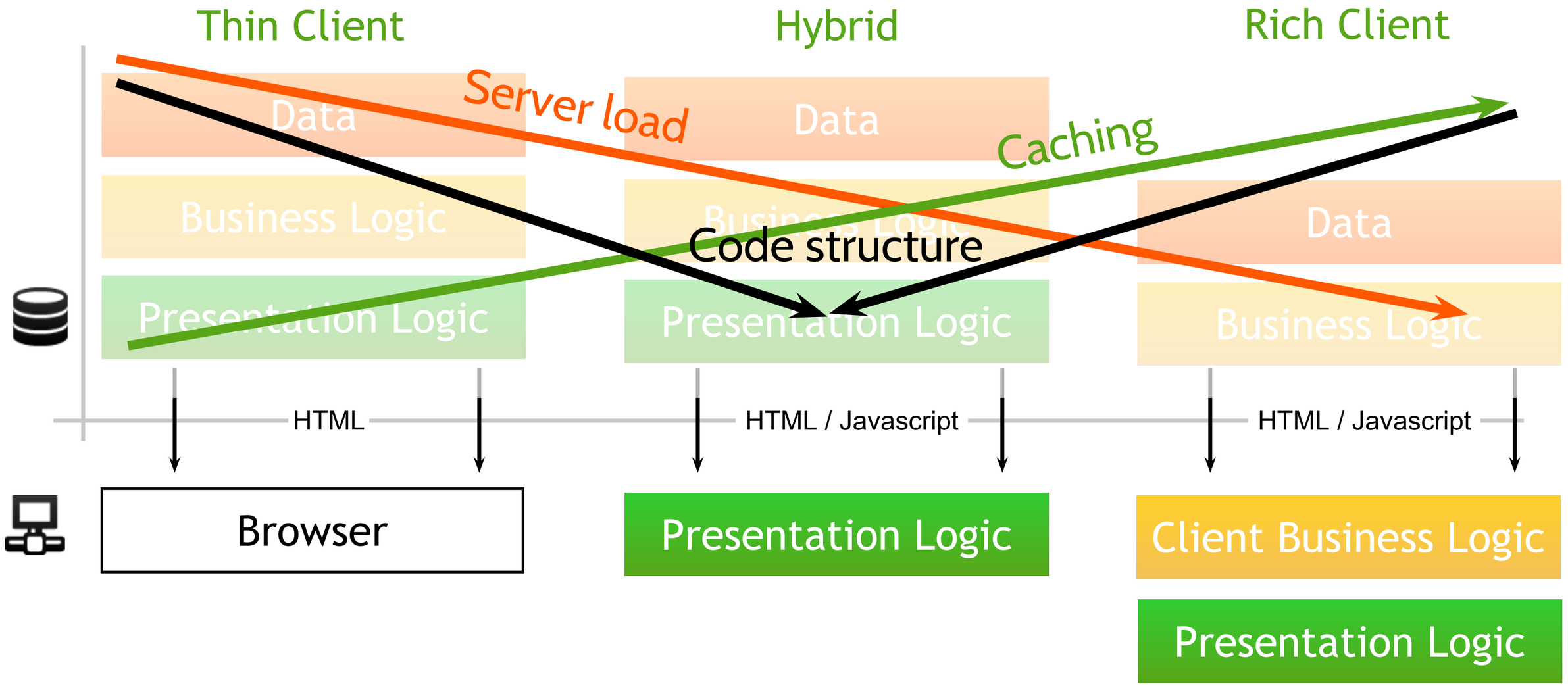
*

Web Applications



*

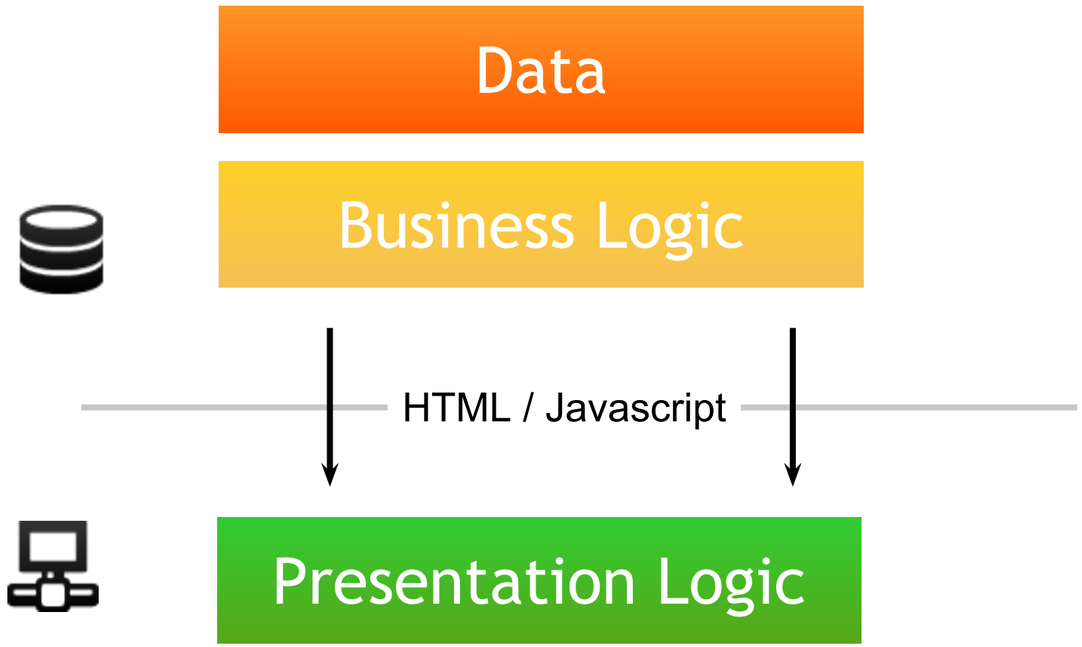
Web Applications



*

Web Applications

Rich Client



RIA Technologies

- ~~Flash / Flex~~
- ~~Silverlight~~
- HTML5 + Javascript

Large Applications

Model-View-Controller

*

<rAppid:js />

rapid rapid development

App of Rich Internet Applications

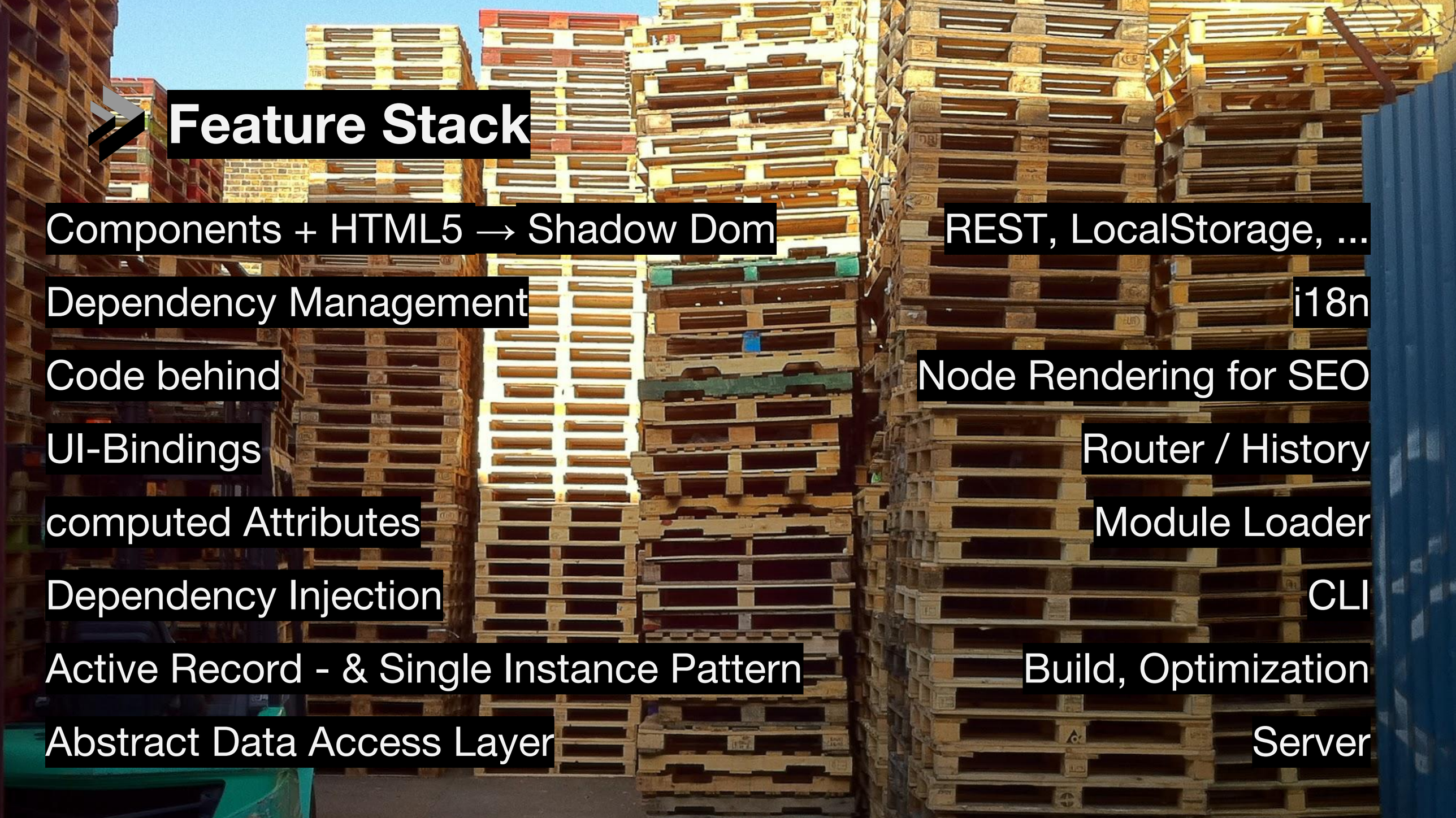
js using Javascript

</> and XML

: with Namespaces

Principles

- Everything is a component
- Use standards (HTML5, JS, XML, CSS)
- Encapsulate complexity
- Updates view via Bindings
- Don‘t repeat yourself
- Produce understandable code, no voodoo



Feature Stack

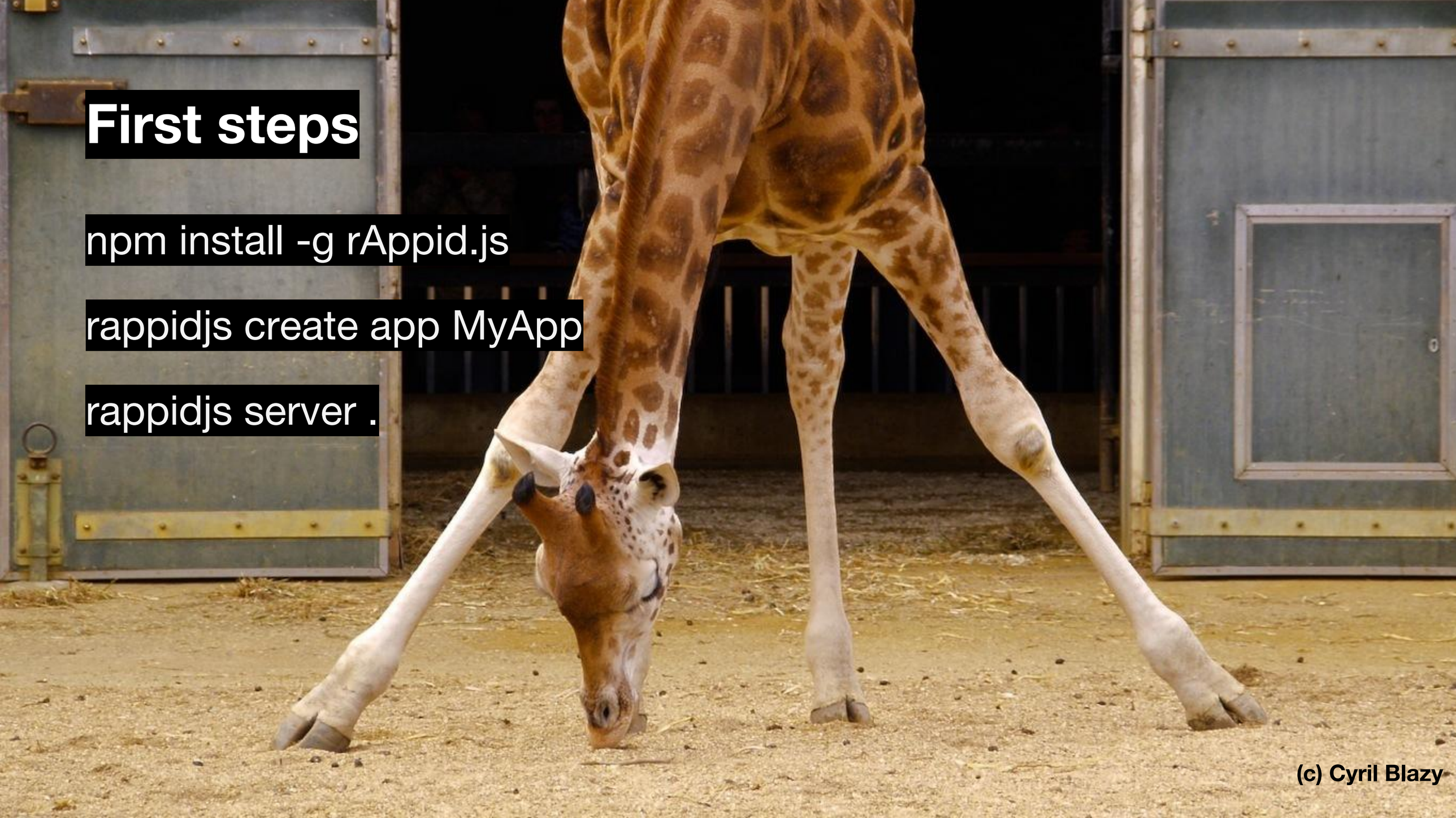
- Components + HTML5 → Shadow Dom
- Dependency Management
- Code behind
- UI-Bindings
- computed Attributes
- Dependency Injection
- Active Record - & Single Instance Pattern
- Abstract Data Access Layer
- REST, LocalStorage, ...
- i18n
- Node Rendering for SEO
- Router / History
- Module Loader
- CLI
- Build, Optimization
- Server

Differences to other MVC frameworks

- XAML instead of Templates

  → no string replacements
- Use XML for UI and configuration
- Advanced binding system

  → only changed parts are re-rendered
- Complete feature stack to write complex apps
- Dependency management (requirejs)
- Runs without jQuery



First steps

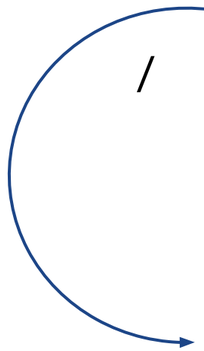
npm install -g rAppid.js

rappidjs create app MyApp

rappidjs server .

Project structure

/ project_dir
- / bin
- / doc
- / **node_modules** // contains npm modules
  - / **rAppid.js** // rAppid.js npm module
    - / **js**
- / **public**
  - / **app**
  - / config.json
  - / index.html
  - / **js**
- / server
  - / web
- / test
- / xsd



/ Addressbook.xml // application start file
/ AddressbookClass.js // code-behind for application
/ collection
/ locale
/ model
/ module
/ view

The index.html

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Sample App</title>

    <script type="text/javascript" src="js/lib/require.js" data-usage="lib"></script>
    <script type="text/javascript" src="js/lib/rAppid.js" data-usage="lib"></script>

    <meta name="fragment" content="!">
```

```
<script type="text/javascript" data-usage="bootstrap">
    rAppid.bootStrap("app/Addressbook.xml");
</script>
```

```
</script>
</body>
</html>
```

*

```
<?xml version="1.0"?>
<app:AddressbookClass xmlns="http://www.w3.org/1999/xhtml"
                      xmlns:js="js.core" xmlns:ui="js.ui" xmlns:app="app">
    <js:Injection>
        <js:I18n locale="en_EN"/>
    </js:Injection>
    <h1>Address Book</h1>

    <h2>New Contact</h2>
    <div>
        <fieldset>
            <label for="firstname">Firstname</label>
            <input id="firstname" type="text" value="{{person.firstname}}"/>

            <label for="lastname">Lastname</label>
            <input id="lastname" type="text" value="{{person.lastname}}"/>

            <label for="phone">Phone</label>
            <input id="phone" type="text" value="{{person.phone}}"/>
        </fieldset>

        <div class="card">
            <p>
                <strong>Name:</strong>
                {person.fullname()}
            </p>
            <p>
                <strong>Phone:</strong>
                {person.phone}
            </p>
```

<Components />

“everything is a component”
component != template

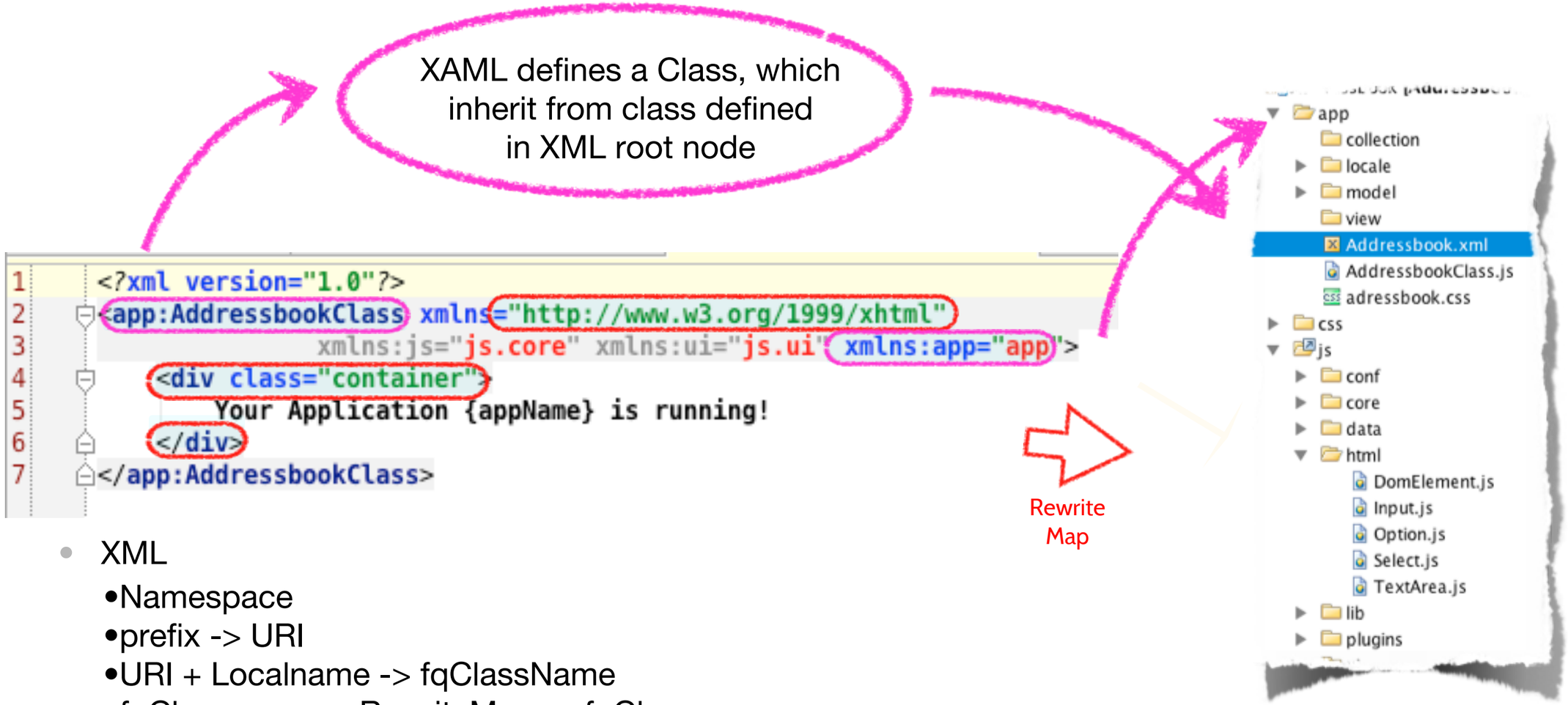
… but what is a component?

Component = A **component** is a software package [...] that encapsulates a set of related functions or data.

Defines:	**structure**	**logic**
	children	*eventhandlers*

*

XAML - eXtensible Application Markup Language



- XML
  - Namespace
  - prefix -> URI
  - URI + Localname -> fqClassName
  - fqClassname -> RewriteMap -> fqClassname

*

XAML Summary

- XAML = **class definition with XML**
- Adressbook.xml **extends** AddressbookClass.js
  - inherits methods
  - inherits vars
- → Accessable from AddressBook

Classes : Prototypes

“single inheritance. No mixins!”

Defining classes

```
define(['app/data/MySuperClass'],function(MySuperClass){
  var someStatic;

  // inherit from Model class
  return MySuperClass.inherit('app.data.MyClass',{
      // class constructor
      ctor: function(){
         // call base constructor
         this.callBase();
      },
      // instance methods
      foo: function(){

      }
  }
});
```

Resolve dependencies

- usage of requirejs

Inherit from MySuperClass

- optional first argument is full qualified classname

call of super method

- without arguments all parameters are passed

Class attributes

- attributes are saved on $ - property

```
1   define(["js/data/Model"], function (Model) {
2       return Model.inherit({
3           defaults: {
4               price: 0,
5               currencySymbol: "€"
6           },
7           resetPrice: function() {
8               this.set("price", 0);
9           },
10          formattedPrice: function () {
11              return this.$.price + " " + this.$.currencySymbol
12          }.onChange('price', 'currencySymbol')
13      });
14  });
```

defaults for **$**

use **this.set** to set values

use **this.$** to read values

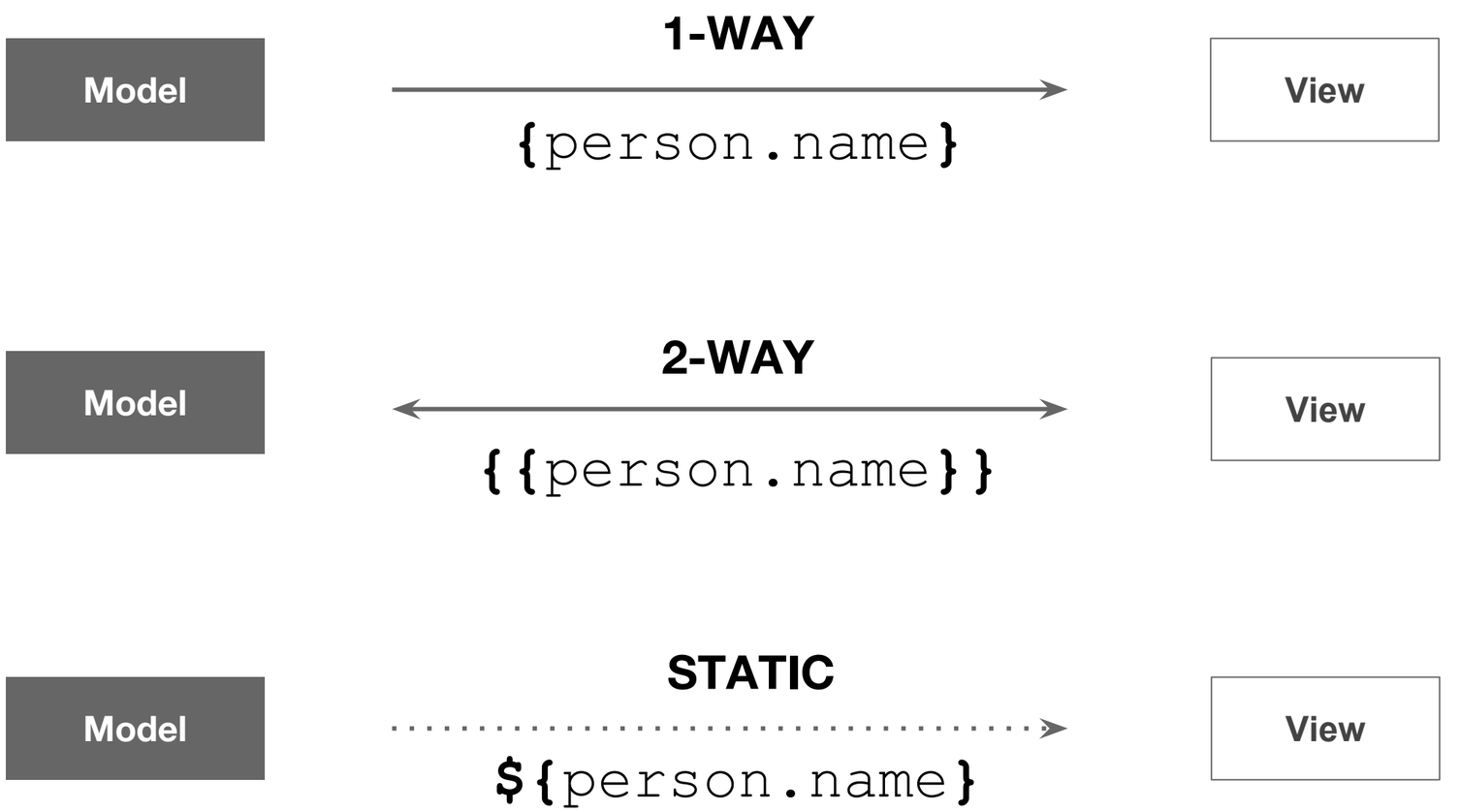
- calculated attributes
  - annotation with onChange(/* fields */) and on(/* events */)

Bindings

“only update the view via bindings”

```
<input type="text" value="{{person.name}}" />
```

Bindings


Model
1-WAY
{person.name}
View
Model
2-WAY
{{person.name}}
View
Model
STATIC
${person.name}
View

Binding paths

```
{person.address.city}
```

```
{person.fullName()}
```

Note: Needs annotation when it changes!

```
{i18n.t('text',person.fullName())}
```

Bindings inside bindings

on=”EventHandler”

“attach to components, not to DOM Elements”

DOM Event Handling

- In XAML

```
<input type="checkbox" onclick="markAllComplete" />

<input type="button" onclick="say('hello', event)" />
```

  - handler defined in Code-Behind

- in Javascript

```
this.bind("on:click", function(e) {
    // event handler function
    alert("hallo");
});
```

*

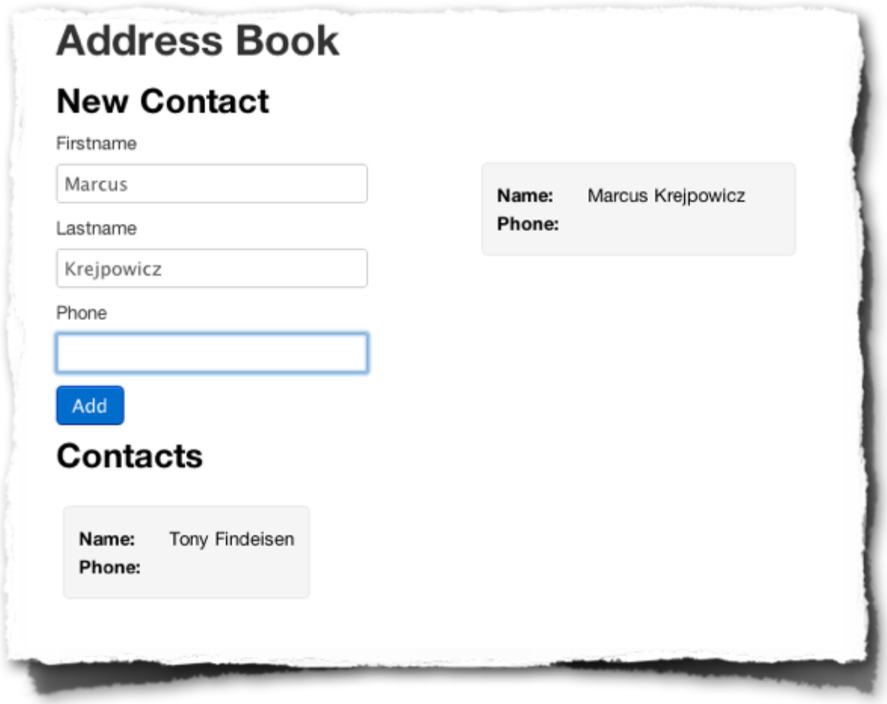
Shadow DOM

“Shadow DOM  *DOM”*

```xml
<?xml version="1.0"?>
<app:AddressbookClass xmlns="http://www.w3.org/1999/xhtml"
                      xmlns:js="js.core" xmlns:ui="js.ui" xmlns:app="app">
    <js:Injection>
        <js:I18n locale="en_EN"/>
    </js:Injection>
    <h1>Address Book</h1>

    <h2>New Contact</h2>
    <div>
        <fieldset>
            <label for="firstname">Firstname</label>
            <input id="firstname" type="text" value="{{person.firstname}}"/>

            <label for="lastname">Lastname</label>
            <input id="lastname" type="text" value="{{person.lastname}}"/>

            <label for="phone">Phone</label>
            <input id="phone" type="text" value="{{person.phone}}"/>
        </fieldset>

        <div class="card">
            <p>
                <strong>Name:</strong>
                {person.fullname()}
            </p>
            <p>
                <strong>Phone:</strong>
                {person.phone}
            </p>
        </div>

        <button onclick="addPerson">Add</button>
    </div>

    <h2>Contacts</h2>
    <ui:ItemsView items="{contacts}" tagName="ul">
        <js:Template name="item">
            <li class="card">
                <p>
                    <strong>Name:</strong>
                    {$item.fullname()}
                </p>
                <p>
                    <strong>Phone:</strong>
                    {$item.phone}
                </p>
            </li>
        </js:Template>
    </ui:ItemsView>

</app:AddressbookClass>
```

Shadow DOM

```
<!DOCTYPE html>
▼<html>
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
    ▼<div>
      <h1>Address Book</h1>
      <h2>New Contact</h2>
      ▼<div>
        ▶<fieldset>…</fieldset>
        ▶<div class="card">…</div>
        <button>Add</button>
      </div>
      <h2>Contacts</h2>
      ▼<ul>
        ▼<li class="card">
          ▶<p>…</p>
          ▶<p>…</p>
        </li>
      </ul>
    </div>
  </body>
</html>
```

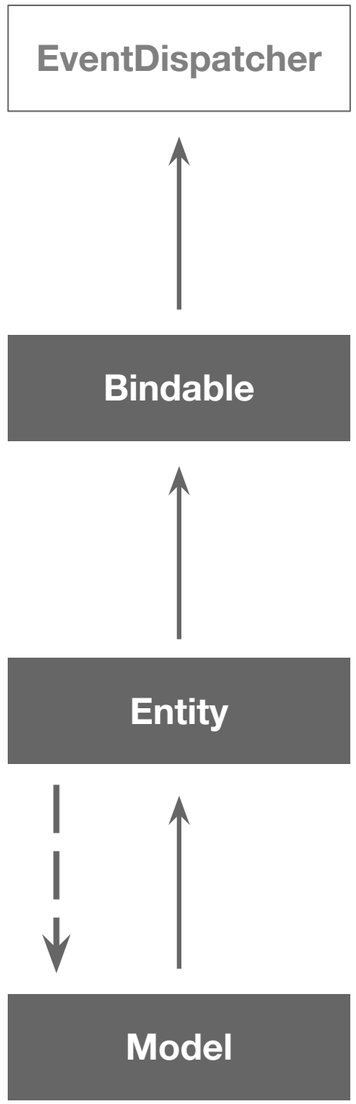


*

MODELS

*

Bindables



Entity cannot live without a Model!

```
defaults: {
      foo: “value”,
      bar: Factory   // new instance of Factory will be created
}
```

```
schema: {
      // schema definition
},
validators: [/* validators goes here */]
```

```
save: function(options, callback) {},
fetch: function(options, callback) {},
remove: function(options, callback) {}
```

Defining a Model

```
define(["js/data/Model", "app/entity/Address"], function(Model, Address){

  return Model.inherit('app.model.Person',{
    schema: {
       name: String,
       birthDate: Date,
       // A dependent address entity
       address: {
         required: false,
         type: Address
       }
    }
  });
});
```

Schema definition

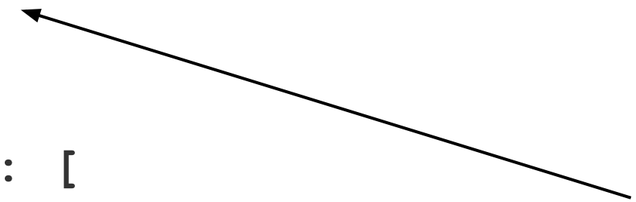
- **field name**
- **type** : Primitive types, prototype factories or paths
- **required**: true | false

Important for Schema validation, data parsing and composing

*

Defining validators in a Model

```
return Model.inherit('app.model.Person',{
      schema: {
          name: String,
          address: Address,
          email: {
              type: String,
              required: false
          }
      },
      validators: [
        new EmailValidator({field: "email"})
      ]
  });
```



Working with models

```
// creates a user model with id 2 in the given data source
var user = this.$.dataSource.createEntity(User,2);

user.fetch({}, function(err, user){
    if(!err){
       alert("user fetched");
    }
});

user.save({}, function(err, user){

})
```

Good morning!

Wednesday

Schedule **Wednesday**

1. Command Line Interface
2. config.json
3. Application Architecture
4. History, Router & ModuleLoader
5. Injection
6. Communication Bus

Commandline interface

```
rappidjs

  help [command]

  create app <AppName>  [dir]   // creates application

  create class app.model.Person js.data.Model // creates class

  config     // writes config.json

  build      // builds application

  doc        // generates docs + schema

  server     // start the server
```

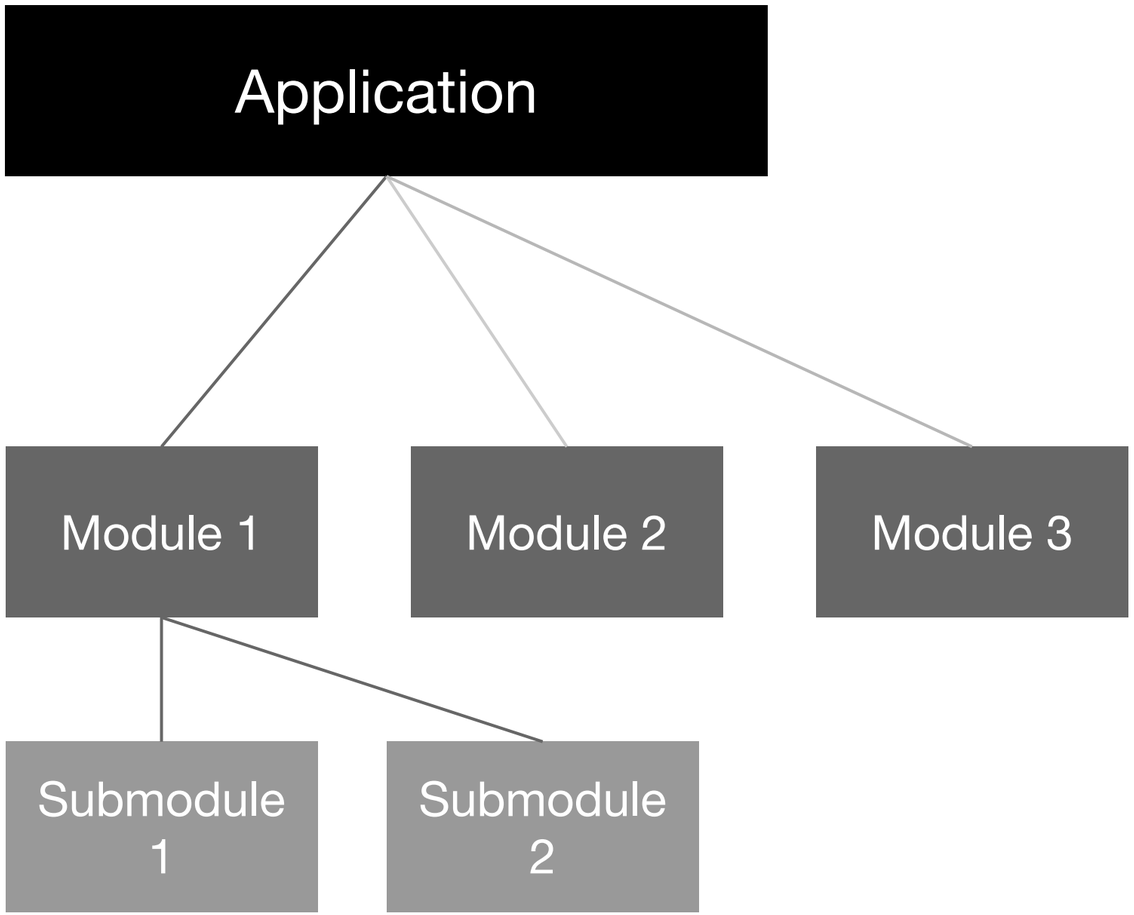
config.json

Contains:

- all XAML classes in directory
- requirejs config
  - shim
  - paths

updated via command `rappidjs config`

Application Architecture



Services

I18n

RestDataSource

...

Router

View Components

DataGrid

TabView

ItemsView

Button

Slider

...

Models + Entities

Configuration

Zone

Address

...

bootstrap

```
rappid.bootstrap(application, target, parameter, config,
  function(err, stage, application) {
      if (!err) {
        // application loaded, initialized & rendered
    }
});
```

application Class : Application *// e.g. app/App.xml*

target DomElement *// document.body*

parameter Object *// {}*

config “config.json” *// loads config file*

|| Object // {baseUrl: “foo/bar”, loadConfiguration: “config.json”}

callback function(err, stage, application)

Codebehind

AppClass.js

```
defaults: {
   /***
    * @codeBehind
    * @type: js.ui.Button
    */
   myComponent: null
}
```

App.xml

```
<app:AppClass>
   …
      <ui:Button cid=”myComponent” />
   …
</app:AppClass>
```

1. define default in code behind

   → rappidjs doc

   → nice IDE support

2. **cid** registers component on **$**

3. bindings are evaluated after component has been initialized

Injection

*

Setup injection

In XAML

```
<js:Injection cid="injection">
    <js:I18n cid="i18n" path="app/locale" locale="en" />
    <conf:Configuration factory="some.class" singleton="true" />
</js:Injection>
```

In Javascript

```
// typed injection
injection.addInstance(someInstance);
// named injection
injection.addInstance('api', someInstance);
```

inject dependencies

SomeCodeBehindClass.js

```
// defaults
defaults: {
    ...
},
// injection
inject: {
    api: "api",
    i18n: I18n,
    configuration: Configuration
}
```

Injectable by default

- Bus
- History
- HeadManager
- ExternalInterface
- InterCommunicationBus
- Environment (“ENV”)
- js.core.ErrorProvider

History

URL parts after # don‘t reload the page

- “HashBang Navigation”
  - `window.onhashchange`
  - `setTimeout` - Fallback for IE <= 7

```
history.navigate(fragment, createHistoryEntry,
triggerRoute, callback);
```

*

Router

1. Checks hash (`#/article/123`)
   - for regular expression **`^article/(\d+)$`**
2. Invokes routing handler function
   - synchronous || asynchronous with **.async()** annotation
   - with routeContext // first parameter

     → supports navigation
     → supports end of routing execution stack
   - parameter from regular expression

*

Router Configuration

```
<js:Router>
    <conf:RouteConfiguration name="default" route="^$" onexec="showAll"/>
    <conf:RouteConfiguration name="show"
          route="^article/(\d+)$" onexec="showArticle"/>
</js:Router>
```

```
showArticle: function (routeContext, articleId) {
    // do something
}.async()  // this route is executed asynchronously
           // invoke routeContext.callback() when done
```

Module Loader

- modularize large applications into modules
- load module asynchronous based on route
  - module can define new routes
  - **re-dispatch** route on loaded module
- ContentPlaceHolder shows content of loaded Module

Module Loader

ModuleConfigurations

Route

Module Classname

Name

+

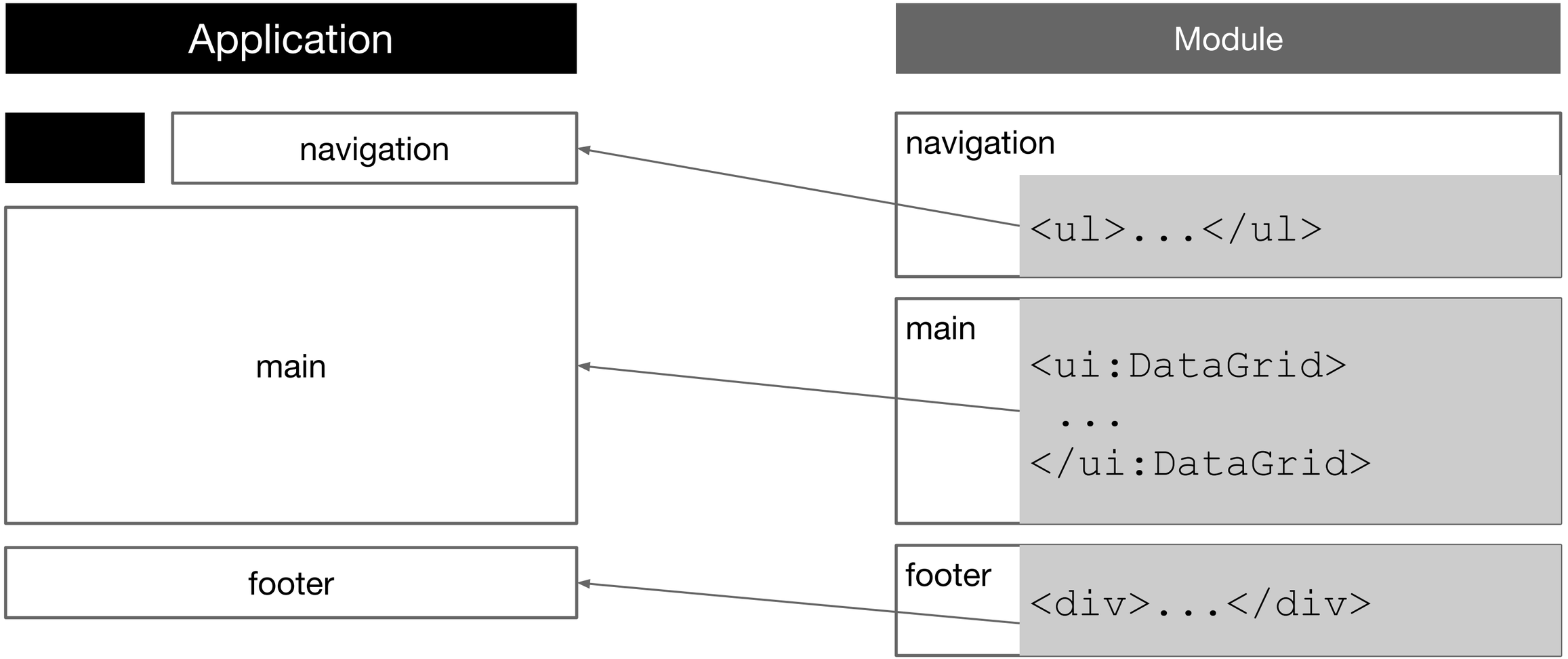
ContentPlaceHolders

Name

Module Configurations

```
<js:ModuleLoader router="{router}">
    <conf:ModuleConfiguration name="articles"
            moduleClass="app.module.Articles" route="^articles.*$"/>
    <conf:ModuleConfiguration name="details"
            moduleClass="app.module.ArticleDetails"
            route="^article\/.*$"/>
    <conf:ModuleConfiguration name="basket"
            moduleClass="app.module.Basket" route="^basket$"/>
...
```

Module Loader - Content Placeholder



Module Loader - Callstack

- **Application**
  - start: function(parameter, callback) // start method of app
  - defaultRoute: function(routeContext) // route callback
- // Module Loader loads module
- **Module1**
  - start: function(callback, routeContext) // start method of module
  - moduleRoute: function(routeContext) // route callback of module

I18n

Attributes:

```
locale: “de_DE” // loads de_DE.json

path: “app/locale”  // dir where locale is
```

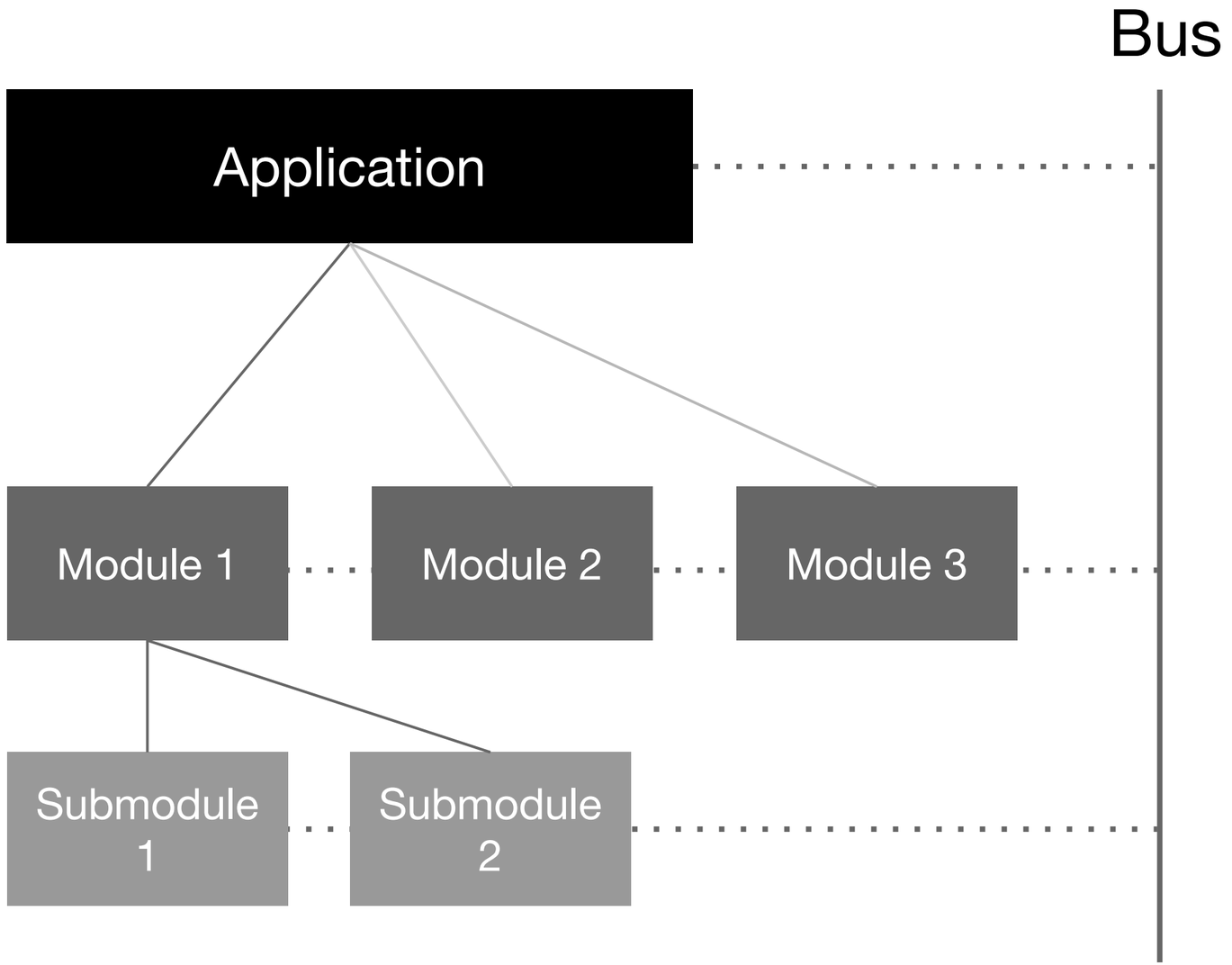
Available Methods

```
i18n.t(key, arg0, arg1)

i18n.ts(keyPart1, keyPart2, arg0, arg1)

i18n.f(date|number, format)
```

I18n - Examples

```
i18n.t(“text”,”foo”,”bar”)

“Hello %0, This is %1”; -> “Hello foo, This is bar”


i18n.t(1, ‘articles’);   -> uses “articles”

i18n.t(4, ‘articles’);  -> uses “articles_plural”

i18n.ts(‘article’,type);-> uses “article.”+type
```

MessageBus


Bus
Application
Module 1
Module 2
Module 3
Submodule 1
Submodule 2

MessageBus

Module 1

```
foo: function(){
  this.$.bus.trigger("Bus.fooEvent", {data: 1});
}
```

Module 3

```
handleFooEvent: function(e){
  var data = e.$;
  // do something on event
}.bus('Bus.fooEvent')
```

Creating Custom components

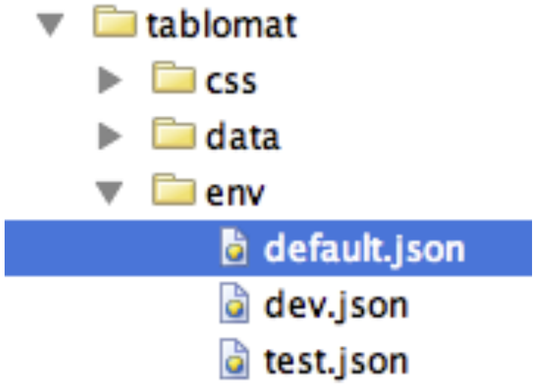
to the IDE

and show TodoView.xml

Environments

AppClass.js

```
supportEnvironments: true,                    // enable environment support

applicationDefaultNamespace: "tablomat",      // search “env” inside

_getEnvironment: function () {  // method which determines environment
     if (/vm1[0-9]{2}\.v/.test(location.hostname)) {
        return "test";
     }
     return "live";
}
```



ExternalInterface

```
define(["js/core/Component", "js/core/ExternalInterface", "js/core/Bus"], function (C, ExternalInterface, Bus)
{
    return C.inherit("app.ExternalApi", {
        inject: {
            externalInterface: ExternalInterface,
            bus: Bus
        },
        initialize: function () {
            this.callBase();

            var externalInterface = this.$.externalInterface;
            externalInterface.addCallback("setProductTypeId", this.setProductTypeId, this);
        },

        setProductTypeId: function (productTypeId, callback) {
          // implementation
        }

    });
});
```

<rAppid:js />

Visit **http://rappidjs.com,** follow **@rappidjs**
or write us an email **support@rappidjs.com**

Hosted on github **rappid/rAppid.js**, but install via npm **rAppid.js**

Developed by Tony Findeisen & Marcus Krejpowicz
licensed under **MIT**

Thank you.

Questions?

Good morning!

Thursday

Schedule **Thursday**

1. Setup development environment, checkout

   --- Breakfast ---

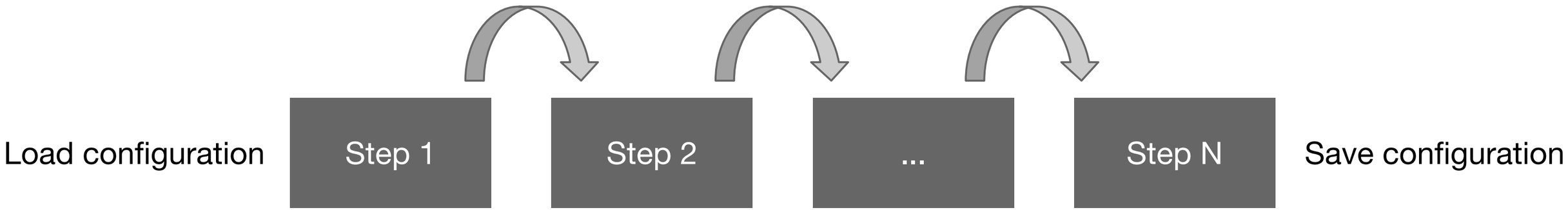
2. Define the data structure, validation
3. Wizard steps, navigation

   --- Lunch ---

4. Modularize application
5. Save configuration
6. build

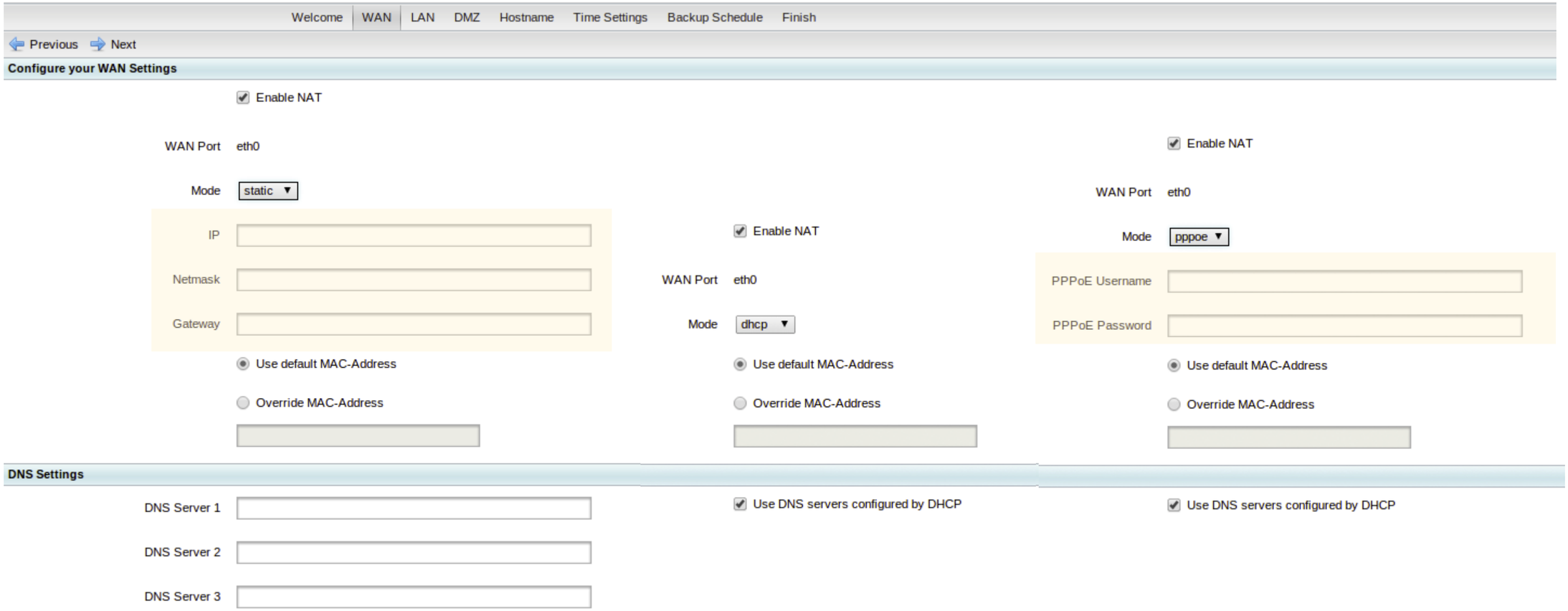
Whizard

validate configuration on each step

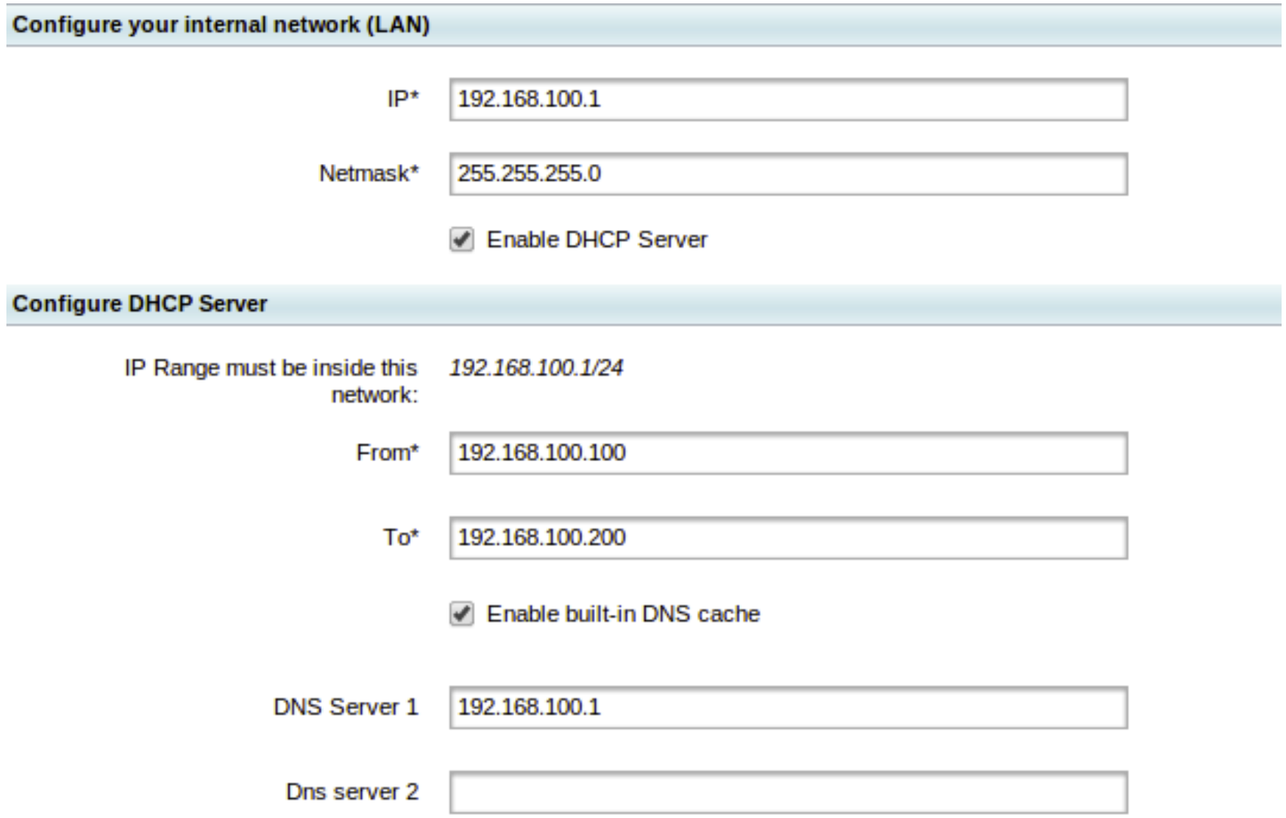


navigation

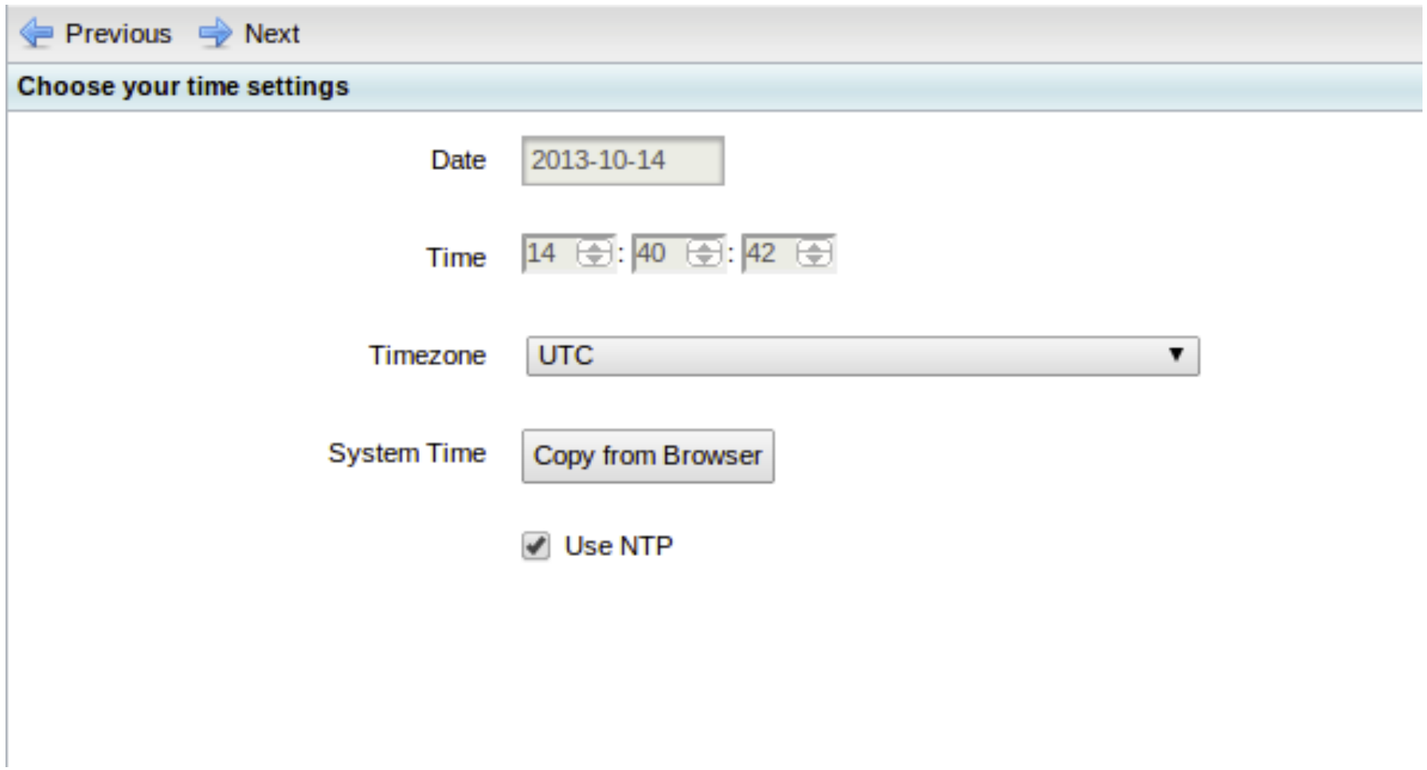
Whizard **WAN**


Welcome
WAN
LAN
DMZ
Hostname
Time Settings
Backup Schedule
Finish
Previous
Next
Configure your WAN Settings
Enable NAT
WAN Port
eth0
Mode
static
IP
Netmask
Gateway
Use default MAC-Address
Override MAC-Address
Enable NAT
WAN Port
eth0
Mode
dhcp
Use default MAC-Address
Override MAC-Address
Enable NAT
WAN Port
eth0
Mode
pppoe
PPPoE Username
PPPoE Password
Use default MAC-Address
Override MAC-Address
DNS Settings
DNS Server 1
DNS Server 2
DNS Server 3
Use DNS servers configured by DHCP
Use DNS servers configured by DHCP

Whizard **LAN**


Configure your internal network (LAN)
IP*
192.168.100.1
Netmask*
255.255.255.0
Enable DHCP Server
Configure DHCP Server
IP Range must be inside this network:
192.168.100.1/24
From*
192.168.100.100
To*
192.168.100.200
Enable built-in DNS cache
DNS Server 1
192.168.100.1
Dns server 2

Whizard **LAN**


Previous
Next
Choose your time settings
Date
2013-10-14
Time
14 : 40 : 42
Timezone
UTC
System Time
Copy from Browser
Use NTP

Breakfast

Thank you for your audience.

Questions?

*

Build

build.json

rappidjs build command

output

summary

first application with rAppid:js

used data binding

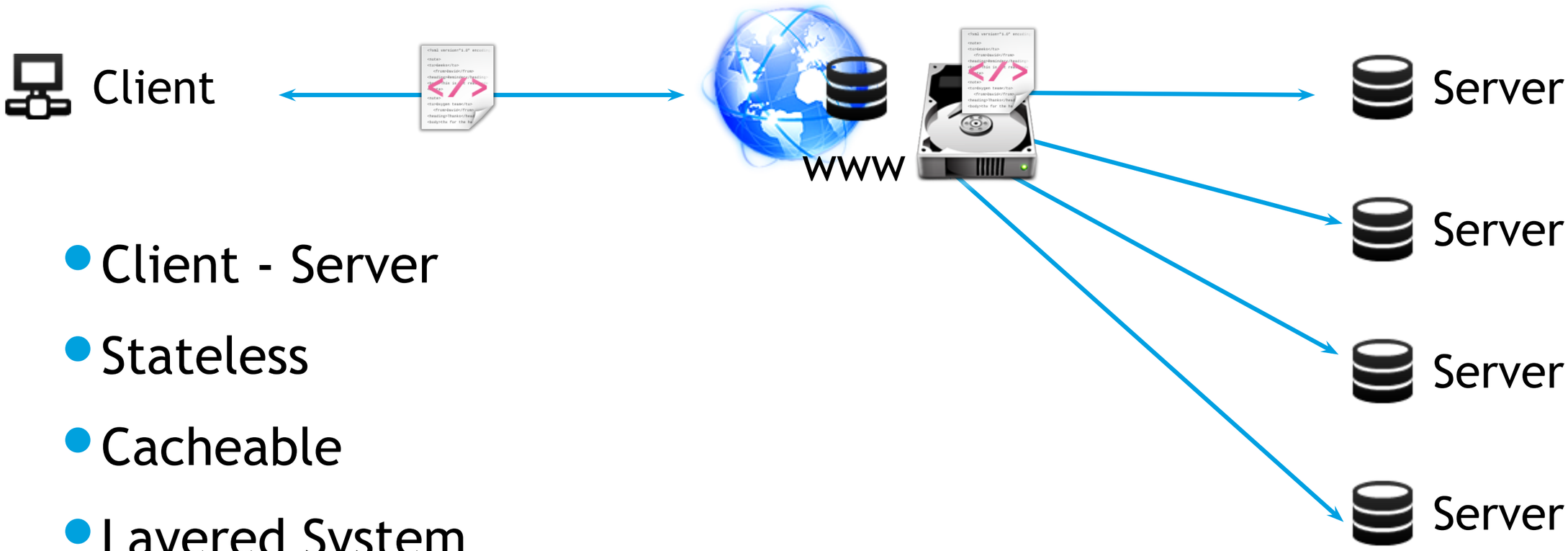
*

Datasource

- Save & Load
  - asynchronous
  - serialize & deserialize
  - context sensitive
- Single-Instance-Cache
- Active Record Pattern

*

REST - Representational state transfer



- Client - Server
- Stateless
- Cacheable
- Layered System
- Representations of Resources (JSON, XML, SVG, ...)

RESTful web services

- Operations via HTTP-Verb

Operation	Collection `api/persons`	Model `/api/persons/1`
GET	List the URIs to contained Models (+ Model data)	Retrieve a representation of the addressed model
POST	Creates a new Model in the Collection -> returns URI	-
PUT	-	Replaces the Model
DELETE	-	Deletes the Model

*

Collections

- Filter via queries
- Paging through offset & limit parameters
- allow creation of new

*

summary

data sources

understanding of REST

*

next steps

How to write complex Rich Internet Applications?

Routing

ModuleLoader

Dependency Injection

i18n

*

summary

top level architecture view of RIAs

next steps

Deployment

*

Deployment

- rappidjs build command
- build.json defines build
  - minifying
  - package modules together

*

Tests

- Unit tests - run on CI-Server
  - mocha - Testrunner
  - chaijs - Assertation library
- Web tests
  - webdriver - control the browser using REST
  - SauceLabs.com - SaaS for hosted browser grid
- E.g. https://travis-ci.org/it-ony/rAppid.js

*

Easy to use component system

<XAML />
HTML5 + Custom Components

```
<ui:MenuButton label="Show" type="info">
    <div>
        Hello Menu Button :)
    </div>
</ui:MenuButton>
```

- Reusable components

  → **Don't Repeat Yourself**
- Good structure
- Easy to maintain

synchronize Model  **View**

{{Bindings}}

- Model-View-Binding: {model.property}
- Two-Way-Binding: {{property}}
- Static Binding: ${a.b.c.d.e.f.g}
- Function binding: {person.name([param])}

Model & **View** always in sync