Securing Single Page Applications

Philippe De Ryck – iMinds-DistriNet, KU Leuven

philippe.deryck@cs.kuleuven.be

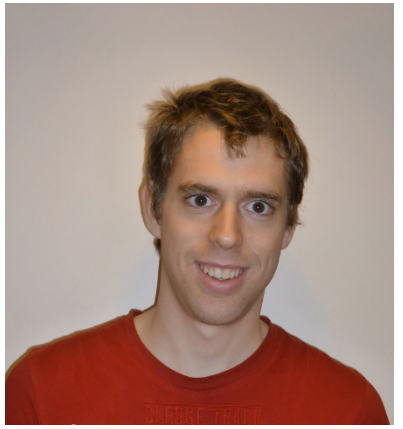
DistriNet iMinds KU LEUVEN

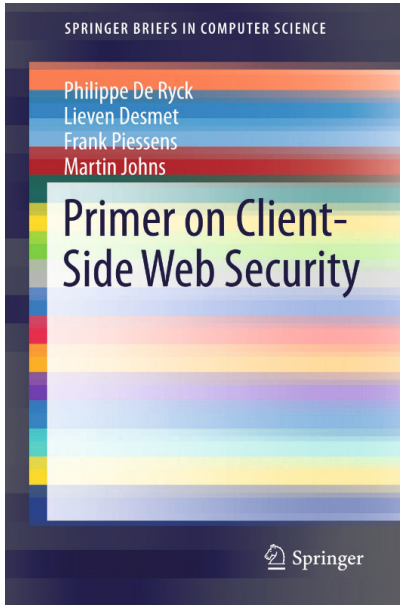
STREWS

SecureApplication Development
secappdev.org

About Me – Philippe De Ryck

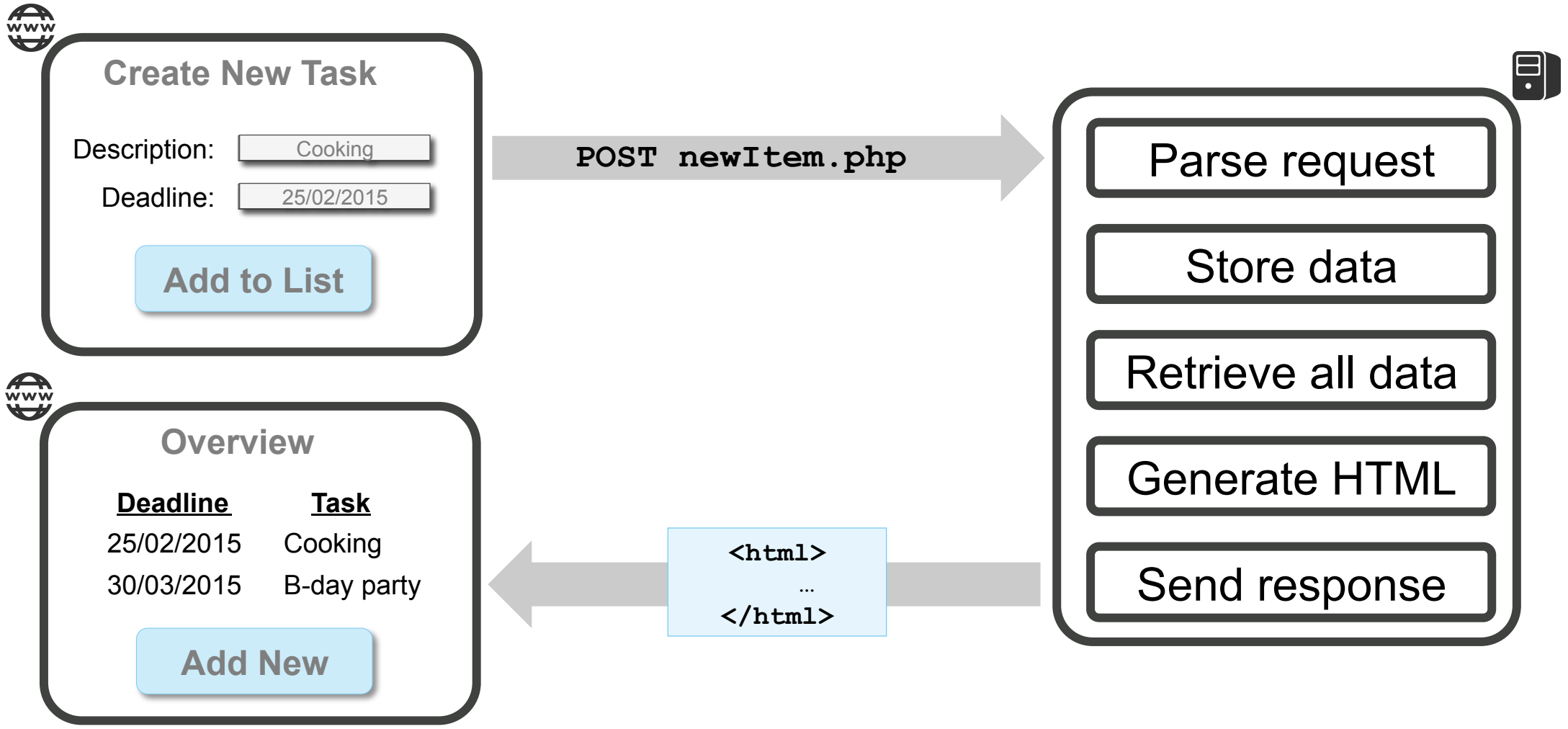
- Postdoctoral Researcher @ DistriNet (KU Leuven)
  - Focus on (client-side) Web security
- Responsible for the Web Security training program
  - Dissemination of knowledge and research results
  - Target audiences include industry and researchers
- Main author of the *Primer on Client-Side Web Security*
  - 7 attacker models, broken down in 10 capabilities
  - 13 attacks and their countermeasures
  - Overview of security best practices



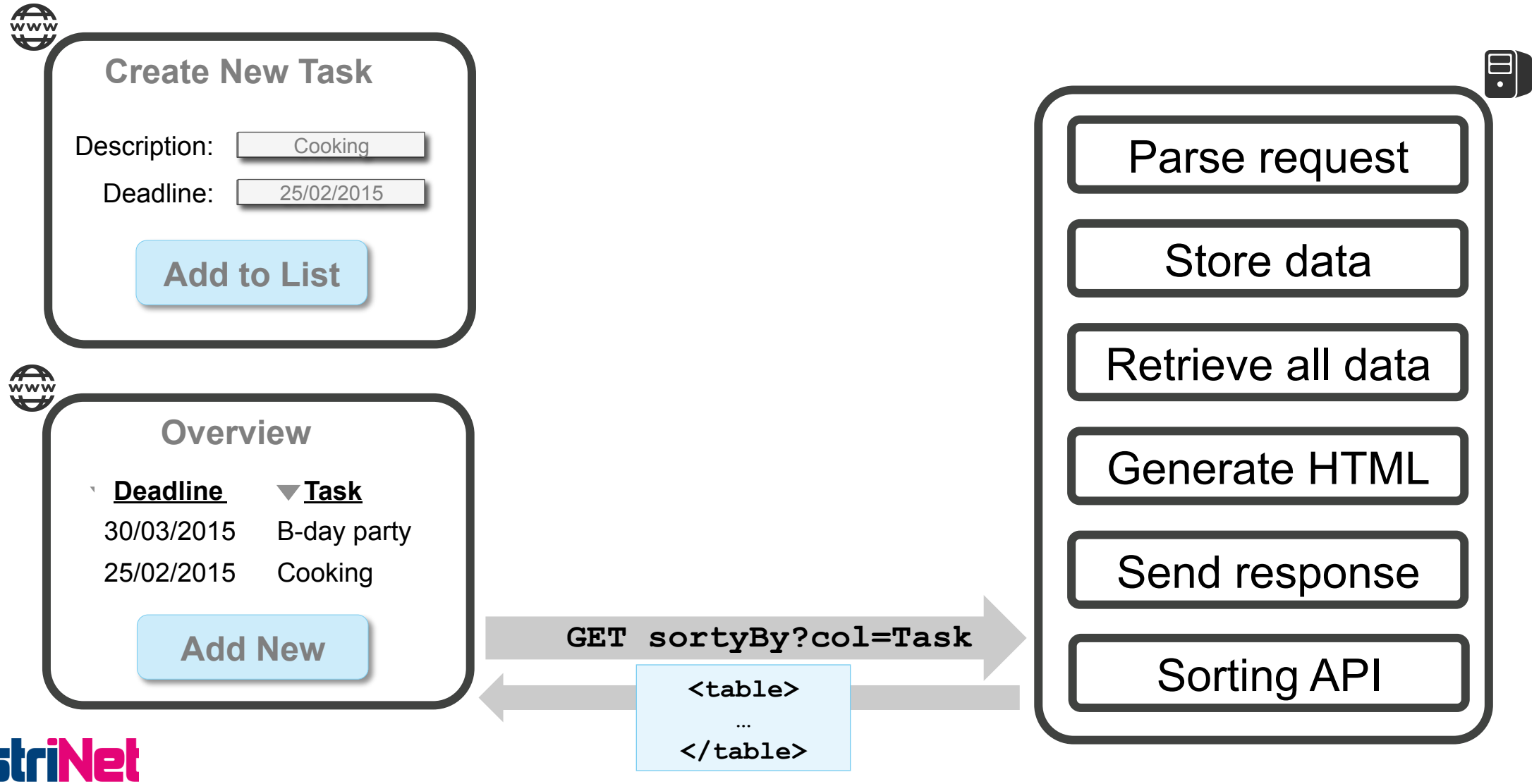


DistriNet iMinds KU LEUVEN STREWS

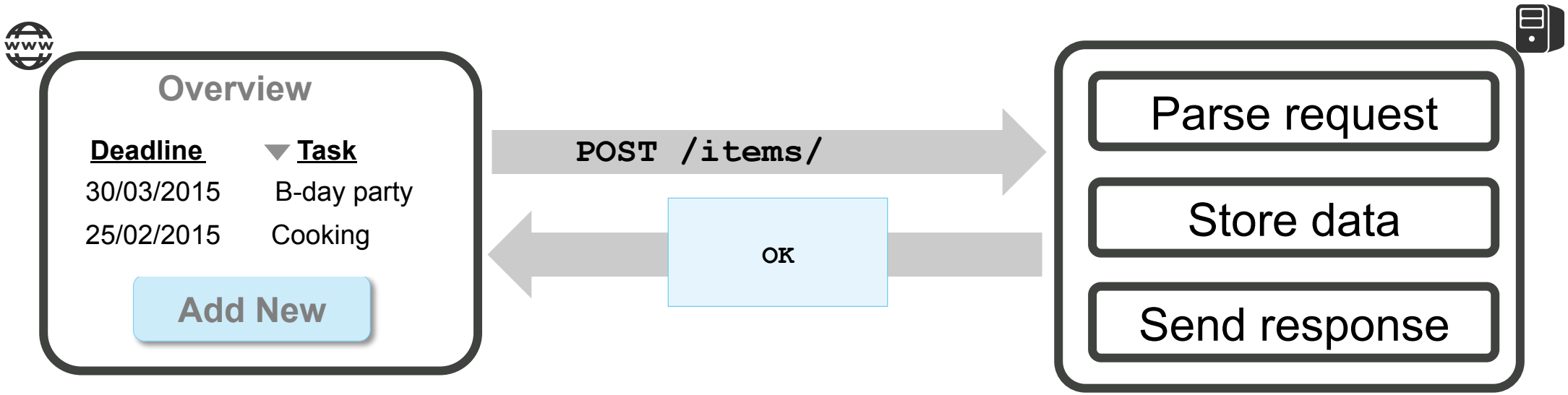
Traditional Web Applications



Traditional Web Applications



Single Page Applications



Outline

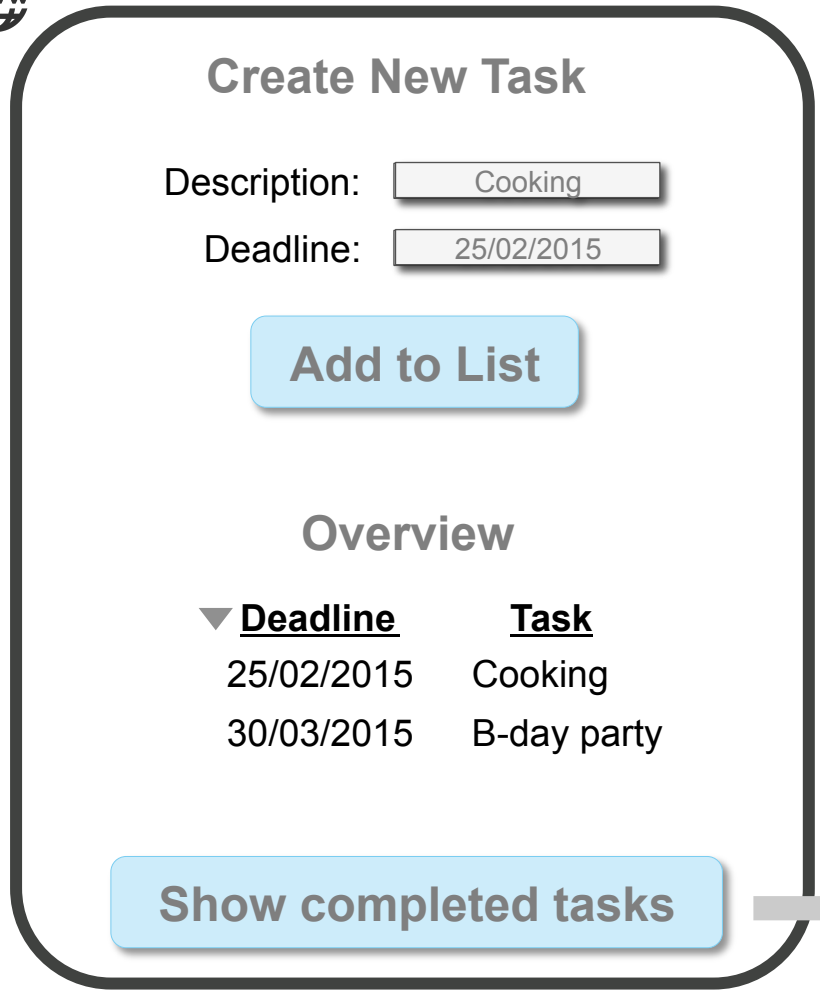
- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Outline

- **The architecture of a single page application**
  - **Moving stuff from the server to the client**
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

What's behind a Single Page Application

`https://items.example.com`



AngularJS routing

```
$routeProvider.when('/overview', {
  templateUrl: 'overview.html',
  controller: 'OverviewCtrl'
}).
$routeProvider.when('/completed',
{
  templateUrl: 'completed.html',
  controller: 'CompletedCtrl'
});
```

`https://items.example.com/#/completed`

What's behind a Single Page Application

https://items.example.com

```
<html ng-app>

  <div ng-controller="NewTaskCtrl">
    …
  </div>

  <div ng-view>
  </div>

</html
```

AngularJS controllers

```
myApp.controller('CompletedCtrl',
  ['$scope', function($scope) {
    $scope.completed = …
}]);
```

AngularJS templates

```
<h3>Completed Tasks</h3>
<ul>
  <li ng-repeat="task in completed">
    {{task.deadline}} {{task.descr}}
  </li>
</ul>
```

What's behind a Single Page Application

- The backend of an SPA has three general responsibilities
  - Serve static application files
  - Provide access to the business logic through an API
  - Persistent data storage
- Frontend and backend are completely decoupled
  - HTTP is the transport mechanism between both
  - RESTful API is a good match for this scenario
- Decoupled backend needs to stand on its own
  - Validate data
  - Enforce workflows

What's behind a Single Page Application

Route	HTTP Verb	Description
/api/bears	`GET`	Get all the bears.
/api/bears	`POST`	Create a bear.
/api/bears/:bear_id	`GET`	Get a single bear.
/api/bears/:bear_id	`PUT`	Update a bear with new info.
/api/bears/:bear_id	`DELETE`	Delete a bear.

https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4

What's behind a Single Page Application

- Properties of a RESTful API
  - Separation of concern between client and server
  - Stateless on the server-side
  - Clear caching decisions (yes or no)
  - Uniform interface

- Many concrete implementations available
  - Heavyweight enterprise frameworks (e.g. Java, .NET)
  - Lightweight JavaScript tools (e.g. NodeJS)
  - Even more lightweight, REST-enabled databases (e.g. CouchDB)

What's behind a Single Page Application

- Consuming a REST API using XHR

Classic XHR

```
var url = "http://…/api/bears/0"
var xhr = new XMLHttpRequest();
xhr.open("DELETE", url, true);
xhr.onreadystatechange = function () {
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            //Bye bye bear 0
        }
    }
}
xhr.send();
```

AngularJS $resource

```
var api = $resource
    ("http://…/api/bears/:id")

api.$delete({id: 0},
  function(v, h) { /* success */ },
  function(res) { /* error */ }
);
```

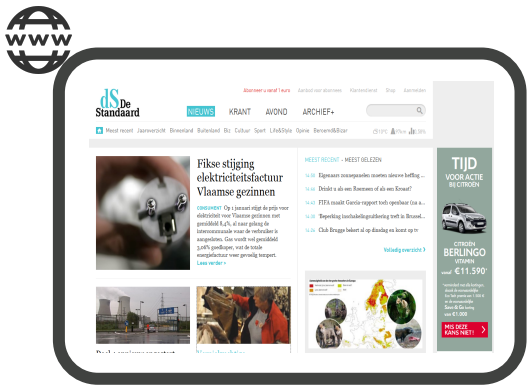
Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- **Authentication and authorization**
  - **In combination with a stateless API**
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Server-Side Session Management



Go to standaard.be

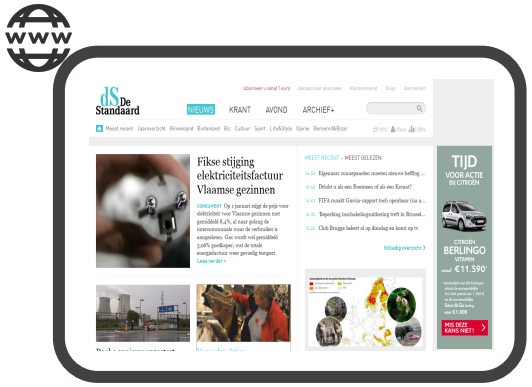
Hello stranger

Login as Philippe

Hello Philippe

Show unread news

Unread news for Philippe



Go to standaard.be

Hello stranger

Login as NotPhilippe

Hello NotPhilippe

standaard.be

3a99a4d1e8f496

Logged_in: false

User: Philippe

2ad3e9f78bc808

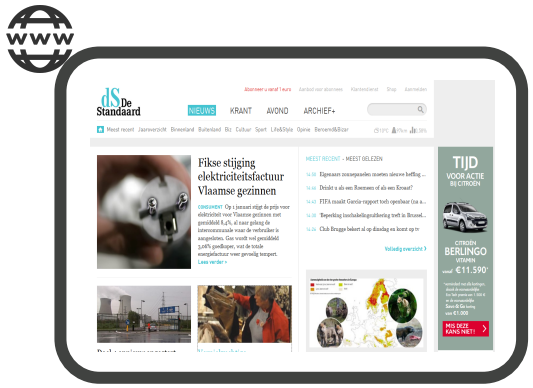
Logged_in: false

User: NotPhilippe

Properties of Server-Side Sessions

- Session information stored at server-side
  - Requires a stateful server-side API
  - Difficult in load balancing scenarios
- Server has full control over sessions
  - Keep track of active sessions
  - Invalidate sessions that have expired
- Session identifier acts as a bearer token
  - Adequate security measures should be in place

Client-Side Session Management



Go to standaard.be

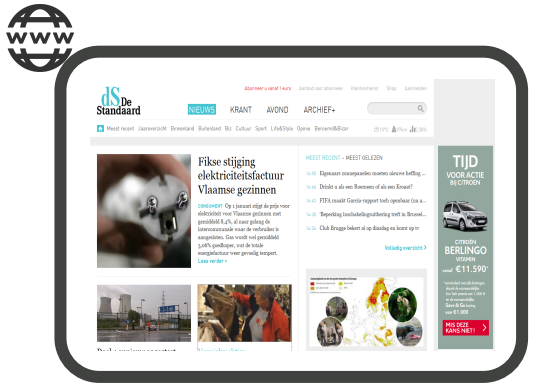
Hello stranger

Login as Philippe

Hello Philippe

Show unread news

Unread news for Philippe



Go to standaard.be

Hello stranger

Login as NotPhilippe

Hello NotPhilippe

standaard.be

Logged_in: false
User: Philippe

Logged_in: false
User: NotPhilippe

Properties of Client-Side Sessions

- Session information generated at server-side
  - But stored within the browser at the client-side
  - Server-side API becomes stateless
- Server loses a level of control over sessions
  - Hard to keep track of active sessions
  - Session expiration requires interaction with the client
- Entire session object is transmitted to the client
  - Also acts as a bearer token, so protect adequately
  - Client can inspect and manipulate session object

Protecting Client-Side Session Data

```
{
    user: Bob,
    isAdmin: false,
    expires: 2015/02/28
}
```

- Store a minimal amount of data
- Prevent client-side manipulation
  - Server signs the session data
  - Server verifies signature when receiving session data
- Prevent continued use of stale sessions
  - Include expiration date in session state and verify when receiving
- If desired, prevent client-side inspection of data
  - Encrypt the session state before sending to the client

Client-Side Sessions – Cookie Example

- Session state stored in traditional cookies
  - State stored as base64 encoded JSON data
  - Server-generated signature stored in additional cookie
  - Browser attaches session state to each request
- Shares all advantages and disadvantages of cookies
  - Available throughout the browser
  - Compatible with cross-origin requests and CORS
    - But can lead to Cross-Site Request Forgery (CSRF)
- Implemented by *cookie-session* for *express* (*Node.JS*)

Client-Side Sessions – JWT Example

- Session state encoded as a JSON Web Token
  - Base64 encoded JSON data
  - Three sections: **header**, **payload** and **signature**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzZWNhcHBkZXYub3JnIiwiZXhwI
joxNDI1MDc4MDAwMDAwLCJuYW1lIjoicGhpbGlwcGUiLCJhZG1pbiI6dHJ1ZX0.uwigNRNPSuH
WXskd1kOd9FmUnnEfa11pEDpVi_Gf06E
```

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

```
{
  "iss": "secappdev.org",
  "exp": 1425078000000,
  "name": "philippe",
  "admin": true
}
```

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

Client-Side Sessions – JWT Example

- Session state encoded as a JSON Web Token
  - Base64 encoded JSON data
  - Three sections: header, payload and signature
  - Currently draft spec at IETF
- Token needs to be attached to every request
  - Configure client-side app to send it in the *Authorization* header
    - Prevents CSRF attacks, but requires CORS preflights
  - Include as part of the request (query parameter, form data)
    - Clutters log files and URLs, but useful for out-of-browser requests
- Numerous libraries available (e.g. *express-jwt*)

Handling Authentication and Authorization

- REST API uses current state for authorization decisions
  - Decision reflected in HTTP response code
    - 200 – OK
    - 401 – Authentication required
    - 403 – Permission denied
- Handle authentication and authorization in client-side app
  - Specify in client-side router when authentication is required
  - Intercept incoming 401 responses and trigger
  - Implement modal login dialog and reroute upon result
    - Clean separation of authentication logic

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- **Injection vulnerabilities and countermeasures**
  - **Getting rid of XSS, if you do it right**
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Traditional XSS Attacks

- Different types of script injection
  - **Persistent**: stored data used in the response
  - **Reflected**: part of the URI used in the response
  - **DOM-based**: data used by client-side scripts

REFLECTED XSS

```
http://www.example.com/search?q=<script>alert('XSS');</script>

        <h1>You searched for<script>alert('XSS');</script></h1>
```

Mitigating XSS in SPAs

- SPA architectures make the client responsible for protection
  - Server only provides data in a specific format
  - Has no idea in which context this data will be used
    - HTML, CSS, JS, …
- Serious MVC frameworks offer good countermeasures
  - AngularJS has *Strict Contextual Escaping*
  - Ember.js does something similar
  - Check your favorite framework for this crucial requirement!

XSS Mitigation in AngularJS

Script

```
x = "javascript:alert(1)"
```

Template

```
<a href="{{x}}"></a>
```

Generated Code

```
<a href="unsafe:javascript:alert(1)"></a>
```

XSS Mitigation in AngularJS

Script

```
x = "<img src='a' onerror='alert(1)'>"
```

Template

```
<div ng-bind="x"></div>
```

Generated Code

```
<div>
    &lt;img src="x" onerror="alert(1)"&gt;
</div>
```

DistriNet iMinds KU LEUVEN

XSS Mitigation in AngularJS

Script

```
x = "<img src='a' onerror='alert(1)'>"
```

Template

```
<div ng-bind-html="x"></div>
```

Generated Code

```
Error: [$sce:unsafe] Attempting to use an
unsafe value in a safe context.
```

DistriNet iMinds KU LEUVEN

XSS Mitigation in AngularJS

Script

```
x = $sanitize("<img src='a' onerror='alert(1)'>")
```

Template

```
<div ng-bind="x"></div>
```

Generated Code

```
<div>
    &lt;img src="x"&gt;
</div>
```

DistriNet iMinds KU LEUVEN

XSS Mitigation in AngularJS

Script

```
x = $sanitize("<img src='a' onerror='alert(1)'>")
```

Template

```
<div ng-bind-html="x"></div>
```

Generated Code

```
<div>
    <img src="x">
</div>
```

DistriNet iMinds KU LEUVEN

XSS Mitigation in AngularJS

Script

```
x = $sce.trustAsHtml("<img src='a'
        onerror='alert(1)'>")
```

Template

```
<div ng-bind="x"></div>
```

Generated Code

```
<div>
    &lt;img src="x" onerror="alert(1)"&gt;
</div>
```

DistriNet iMinds KU LEUVEN

XSS Mitigation in AngularJS

Script

```
x = $sce.trustAsHtml("<img src='a'
        onerror='alert(1)'>")
```

Template

```
<div ng-bind-html="x"></div>
```

Generated Code

```
<div>
    <img src="x" onerror="alert(1)">;
</div>
```

DistriNet iMinds KU LEUVEN

XSS Mitigation in AngularJS

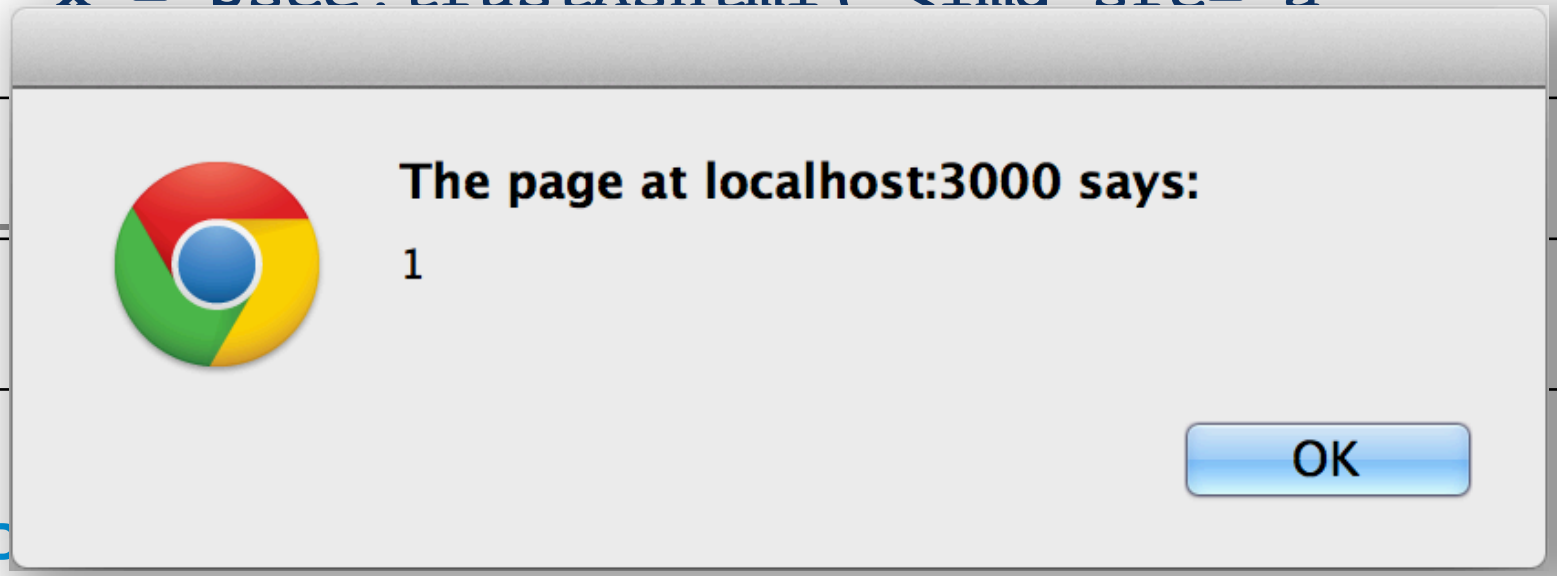
SCRIPT

```
x = $sce.trustAsHtml("<img src='a'
```

TEMPLATE

GENERATED

```
<div>
    <img src="x" onerror="alert(1)”>;
</div>
```



DistriNet
iMinds
KU LEUVEN

How SPA Frameworks Change the Game

- JavaScript MVC frameworks change how the DOM works
  - Extensions through elements, attributes, etc.
  - New interfaces
  - Often in combination with templating

```
<graph class="visitor-graph">
  <axis position="left"></axis>
  <axis position="bottom"></axis>
  <line name="typical-week" line-data="model.series.typicalWeek"></line>
  <line name="this-week" line-data="model.series.thisWeek"></line>
  <line name="last-week" line-data="model.series.lastWeek"></line>
</graph>
```

How SPA Frameworks Change the Game

- JavaScript MVC frameworks change how the DOM works
  - Extensions through elements, attributes, etc.
  - New interfaces
  - Often in combination with templating

- This behavior is enabled by framework processing in JS
  - Highly dependent on *String-to-Code* capabilities
  - Most common examples: *eval()* and the *Function* constructor

- But isn't *eval()* evil?

Mustache Security

- Project dedicated to JS MVC security pitfalls
  - Assuming there is an injection vector
  - Assuming there is conventional XSS filtering in place
  - What can an attacker do?

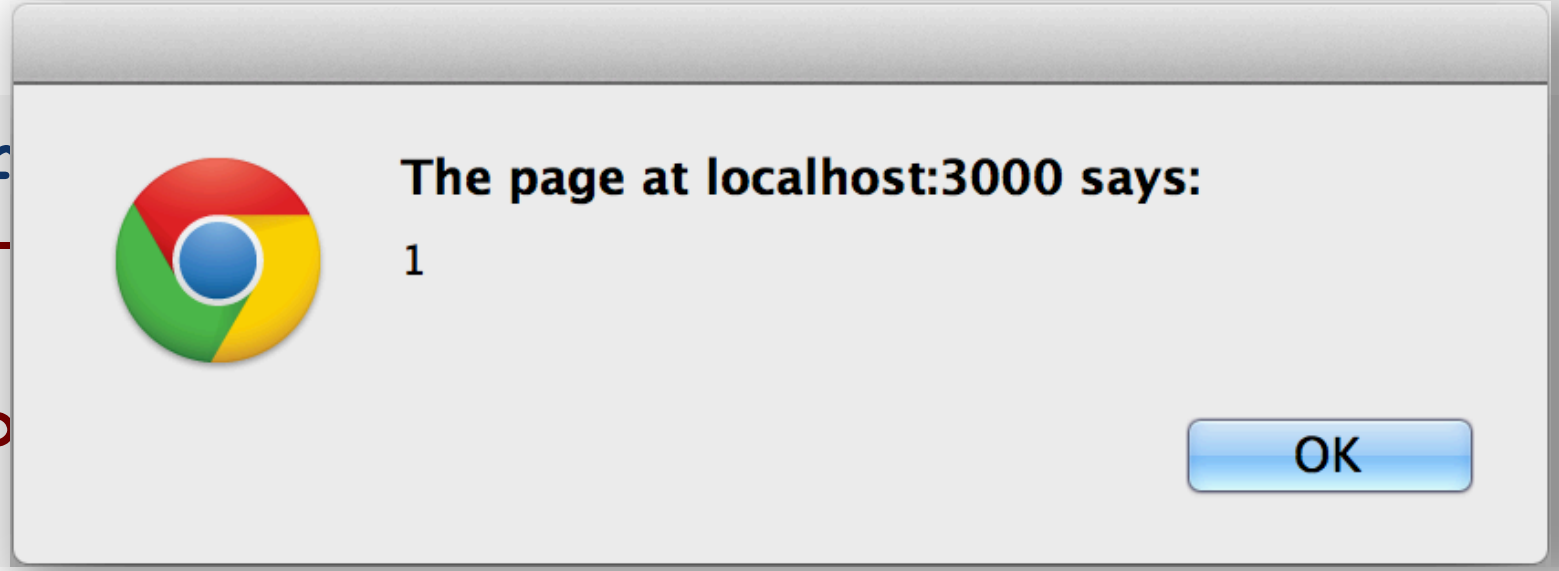
- New behavior often breaks existing security assumptions
  - Bypass currently used security mechanisms
  - Script injection possible whenever a data attribute is allowed

Mustache Security Examples

```
<script src="knockout-2.3.0.js"></script>
<div data-bind="x:alert(1)" />
<script>
        ko.applyBindings();
</script>
```

Mustache Security Examples

```
<script sr
<div data-
<script>
        ko
</script>
```

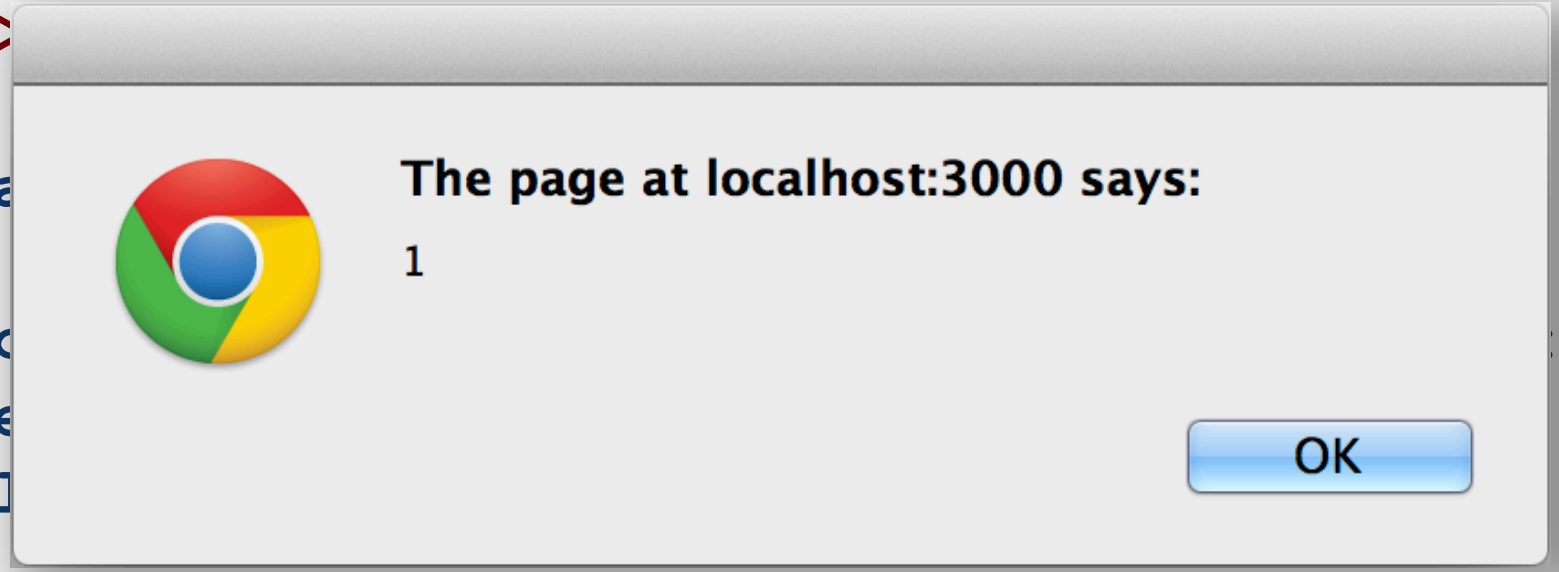


Mustache Security Examples

```
<script src="jquery-1.7.1.min.js"></script>
<script src="kendo.all.min.js"></script>
<div id="x"># alert(1) #</div>
<script>
  var template = kendo.template($("#x").html());
  var tasks = [{ id: 1}];
  var dataSource = new kendo.data.DataSource({ data: tasks });
  dataSource.bind("change", function(e) {
    var html = kendo.render(template, this.view());
  });
  dataSource.read();
</script>
```

Mustache Security Examples

```
<script src="jquery-1.7.1.min.js"></script>
<script src="kendo.all.min.js"></script>
<div id="x">
<script>
  var templa
  var tasks
  var dataSo                                          tasks });
  dataSource
    var html
  });
  dataSource.read();
</script>
```

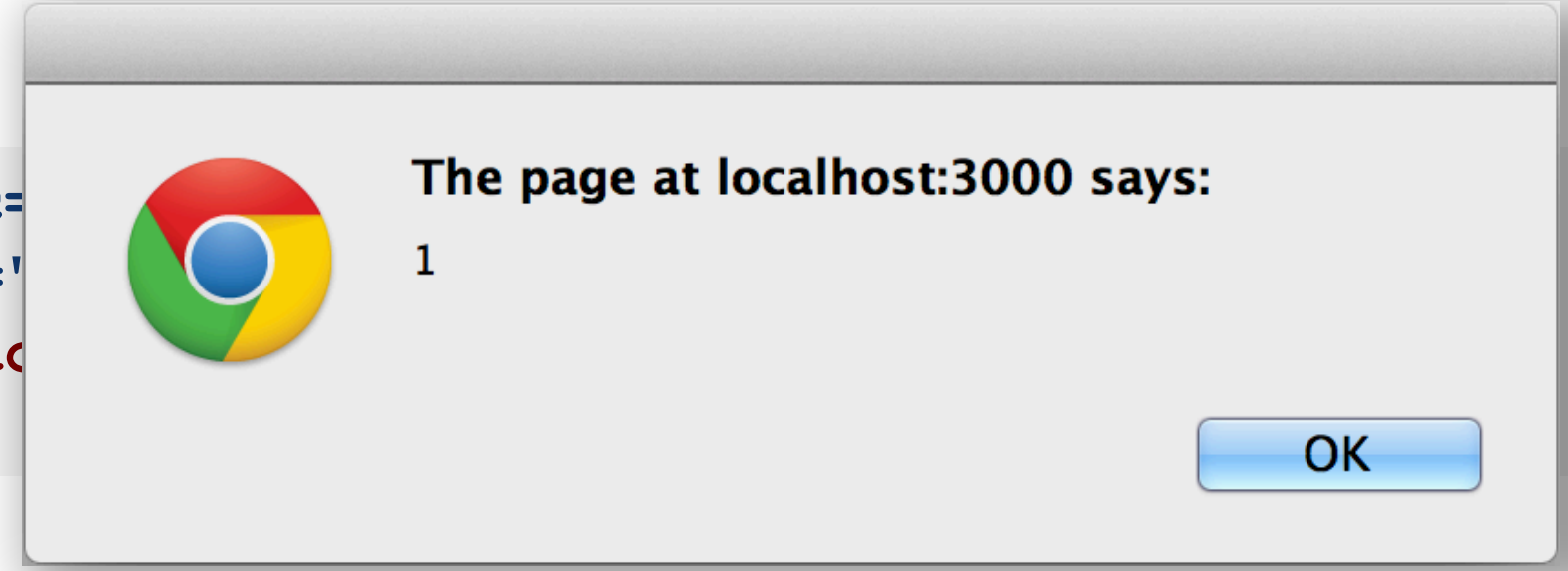


Mustache Security Examples

```
<script src="angular1.1.5.min.js"></script>
<div class="ng-app">
{{constructor.constructor('alert(1)')()}}
</div>
```

Mustache Security Examples

```
<script src=
<div class="
{{constructo
</div>
```



Content Security Policy

- Declares content restrictions on Web resources
  - Specifies allowed sources of included content
    - Images, scripts, frames, …
  - Specifies allowed destinations for certain actions
    - Forms, XHR, WebSockets, …
  - Disables inline scripts and styles by default

INLINE SCRIPTS

```
Content-Security-Policy: default-src: 'self';

http://www.example.com/search?q=<script>alert('XSS');</script>

<h1>You searched for<script>alert('XSS');</script></h1>
```

Content Security Policy

- Declares content restrictions on Web resources
  - Specifies allowed sources of included content
    - Images, scripts, frames, …
  - Specifies allowed destinations for certain actions
    - Forms, XHR, WebSockets, …
  - Disables inline scripts and styles by default
  - Prevents the use of *string-to-code* functionality

- CSP is meant to be a second line of defense against XSS

CSP and JS MVC Frameworks

- Default behavior of MVC frameworks is not CSP compatible
  - Dependent on *string-to-code* functionality
  - Requires *unsafe-eval* in CSP, which kind of misses the point

- Some frameworks offer a special CSP mode
  - AngularJS can easily be made CSP compliant
  - Ember.js templates can be compiled to be CSP compliant
  - Other frameworks provide addons or custom binding providers

ng-csp

```
<html ng-app ng-csp> … </html>
```

- CSP prevents inline scripts from running …

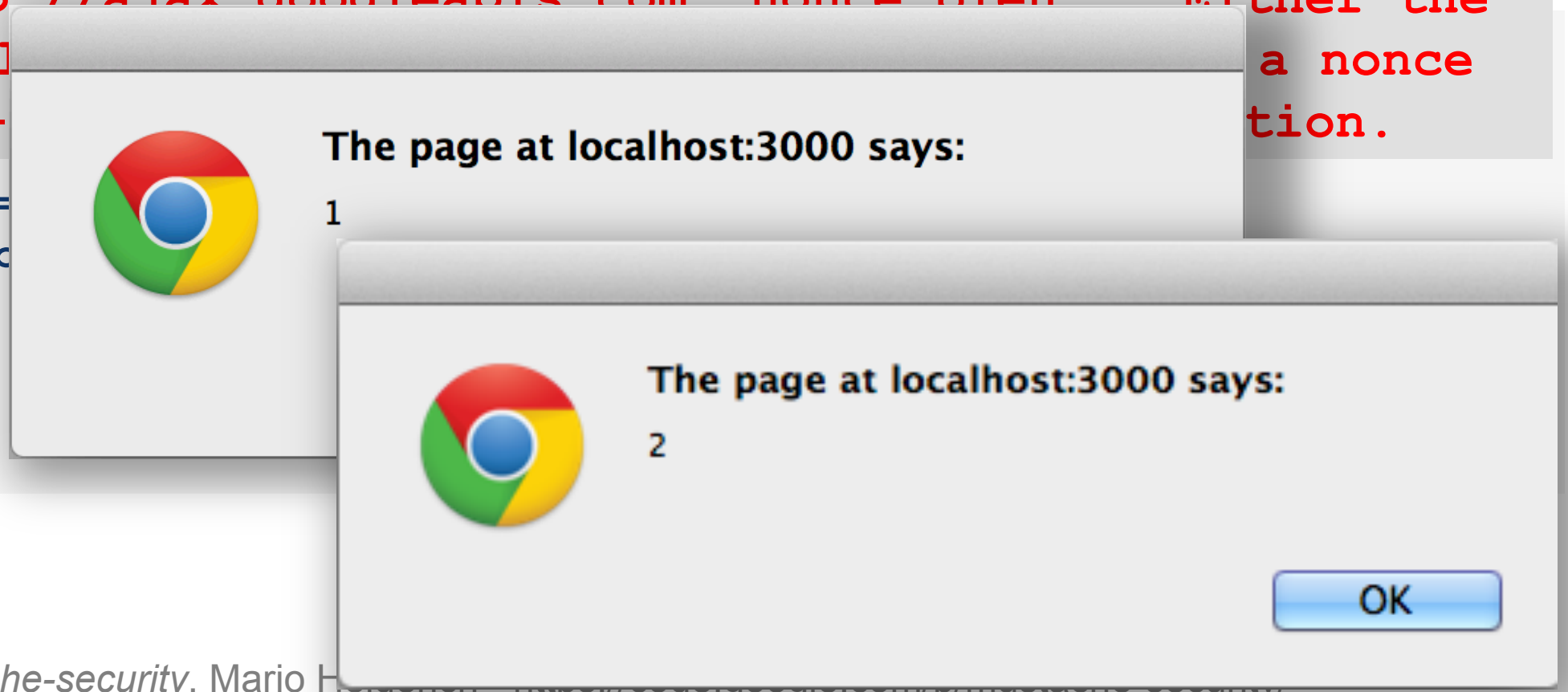
```
<html ng-app ng-csp>
 <body ng-controller="MyController">
  <h1 onclick="alert(0)">Click me</h1>
  <h1 ng-click="$event.view.alert(1)">Click me</h1>
  <h1 ng-mouseover="$event.target.ownerDocument.defaultView.alert(2)”>
      Hover me
  </h1>
 </body>
</html>
```

ng-csp

```
<html ng-app ng-csp> … </html>
```

```
Refused to execute inline event handler because it violates
the following Content Security Policy directive: "script-src
'self' http://ajax.googleapis.com 'nonce-bleh'". Either the
'unsafe-inl                                          a nonce
   ('nonce-                                          tion.
```

```
  <h1 ng-click=
  <h1 ng-mousec
      Hover me
  </h1>
 </body>
</html>
```



ng-csp

- So how does angular process event handlers?
  - Parse ‘ng’-attributes
  - Create anonymous functions, connected with events
  - Wait for event handler to fire

```
$element.onclick = function($event) {
  $event['view']['alert']('1')
}
```

  - Technically, **not inline**, and no **eval()**
- CSP 1.2.x has a strong sandbox
  - No more references to dangerous objects (e.g. *window*)

The Importance of CSP Compliance

- CSP is promising, but hard to apply in legacy applications
  - Adoption on new applications is slowly rising
- Google strongly pushes CSP adoption
  - Chrome extensions must use CSP
    - And *unsafe-inline* will have no effect (*unsafe-eval* can be used)
  - Chrome packaged apps must use CSP
    - Default policy can not be relaxed
    - Only local content is allowed (except media files)
    - Outgoing connections are allowed

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- **Remote API access**
  - **Unintentional through CSRF, and intentional using CORS**

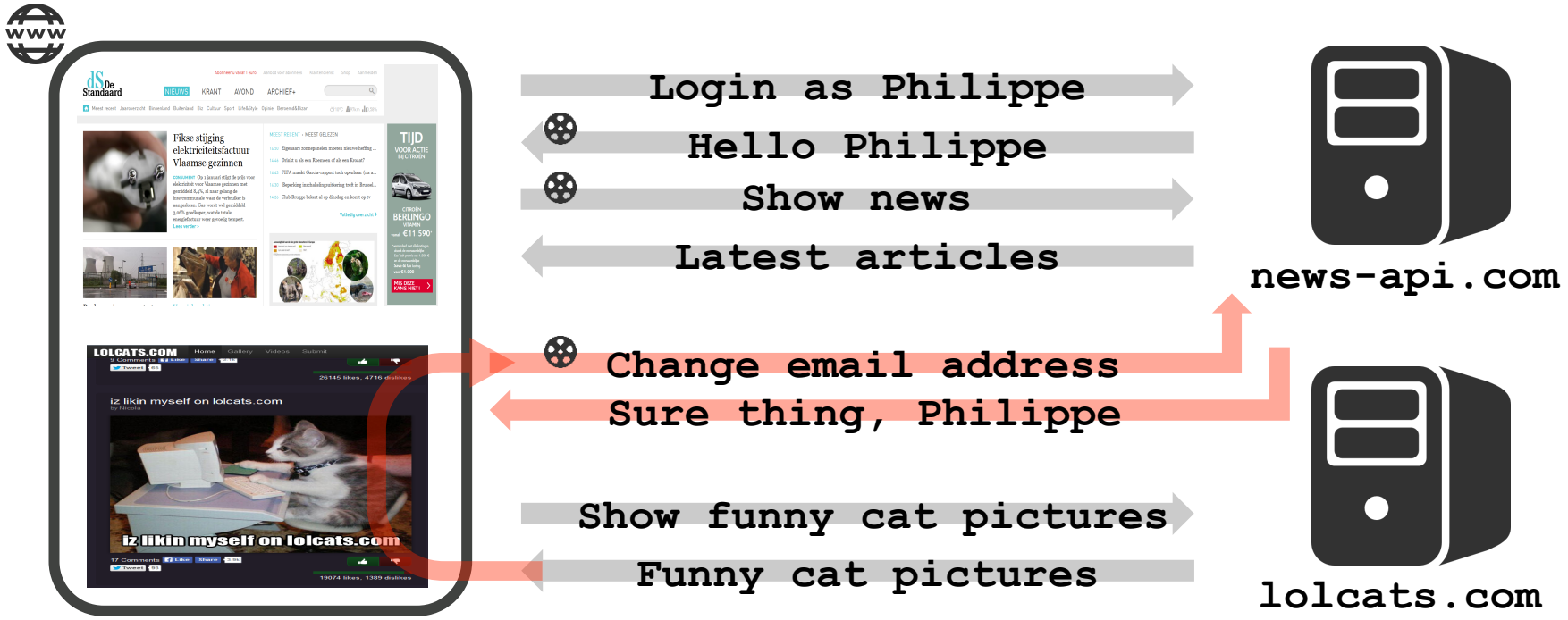
Remote API Access

- But isn't every access remote access?
  - Yes, but the answer lies in the origins
  - Same-origin access has never been restricted
  - Cross-origin access is more interesting

- Securing an API
  - Preventing unintentional cross-origin access through CSRF
  - Enabling intentional cross-origin access

Cross-Site Request Forgery

- Attacker is able to execute requests in the victim's session
- Side-effect of ambient authority of session cookies



Mitigating Cross-Site Request Forgery

- Mitigation techniques need to be explicitly present
  - Token-based approaches
  - Origin header

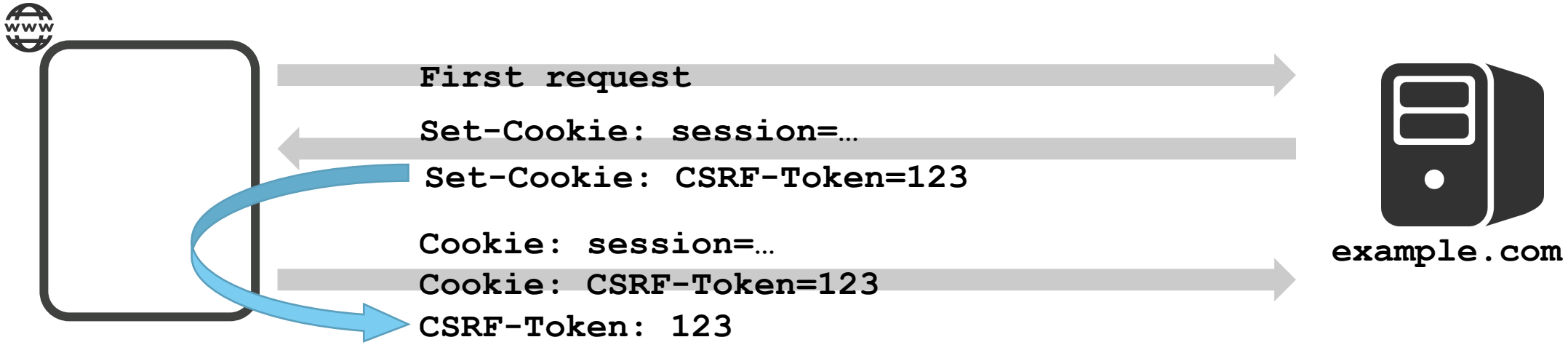
TOKEN-BASED APPROACH

example.com

```
<form action="submit.php">
  <input type="hidden" name="token"
    value="qasfj8j12adsjadu2223" />
  …
</form>
```

Transparent CSRF Tokens

- Hidden tokens strongly depend on HTML
  - Less compatible with JavaScript code
- Solution: transparent tokens using cookies and headers



Only the application can copy cookie value into header

Enabling CSRF Protection

Express

```
var csrf = require('csurf');
app.use(csrf());
app.use("/", function(req, res, next) {
  res.cookie('XSRF-TOKEN', req.csrfToken());
  next();
});
```

AngularJS

```
Enabled by default if cookie is present!
```

Legitimate API Access

- How to get script-based access to cross-origin APIs

```
XMLHttpRequest cannot load http://127.0.0.1:3000/. No
'Access-Control-Allow-Origin' header is present on the
requested resource. Origin 'http://localhost:3000' is
therefore not allowed access.
```

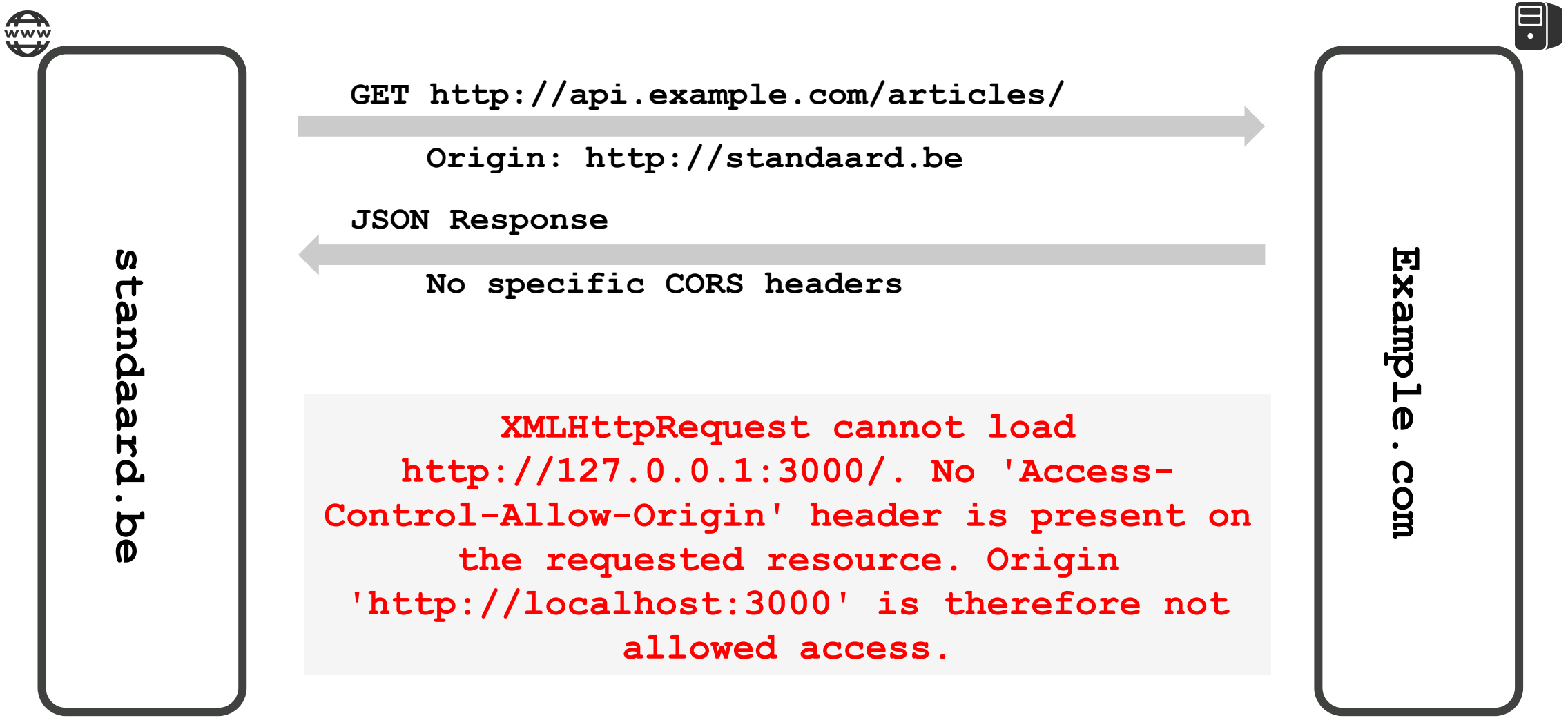
- CORS to the rescue
  - Cross-Origin Resource Sharing (W3C Recommendation)
  - Introduce additional security headers to enable cross-origin XHR
  - Headers ensure that existing security assumptions are not broken

Legitimate API Access

APIs that support CORS

- Amazon S3
- DBpedia Spotlight
- Dropbox API
- Facebook Graph API
- Flickr API
- FourSquare API
- Google APIs
- Google Cloud Storage
- GitHub v3 API
- MediaWiki API
- prefix.cc
- PublishMyData
- sameAs
- SoundCloud API
- Spotify Lookup API
- Sunlight Congress API
- URIBurner
- YouTube API (blog post)
- doctape API

Brief Overview of CORS



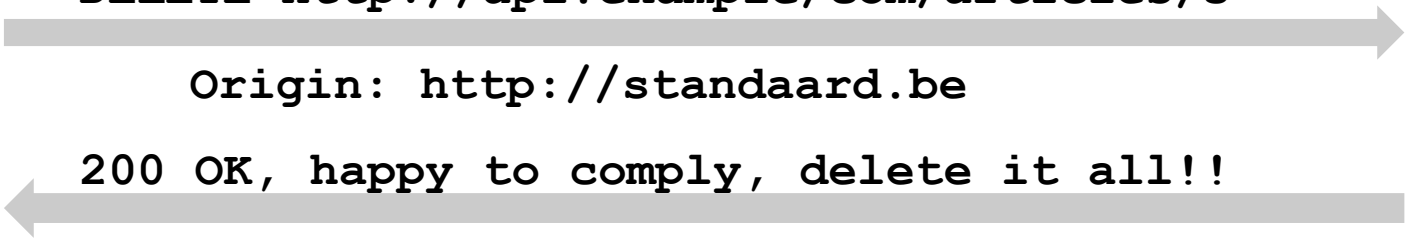

```
XMLHttpRequest cannot load
http://127.0.0.1:3000/. No 'Access-
Control-Allow-Origin' header is present on
the requested resource. Origin
'http://localhost:3000' is therefore not
allowed access.
```

Brief Overview of CORS

Brief Overview of CORS

standaard.be

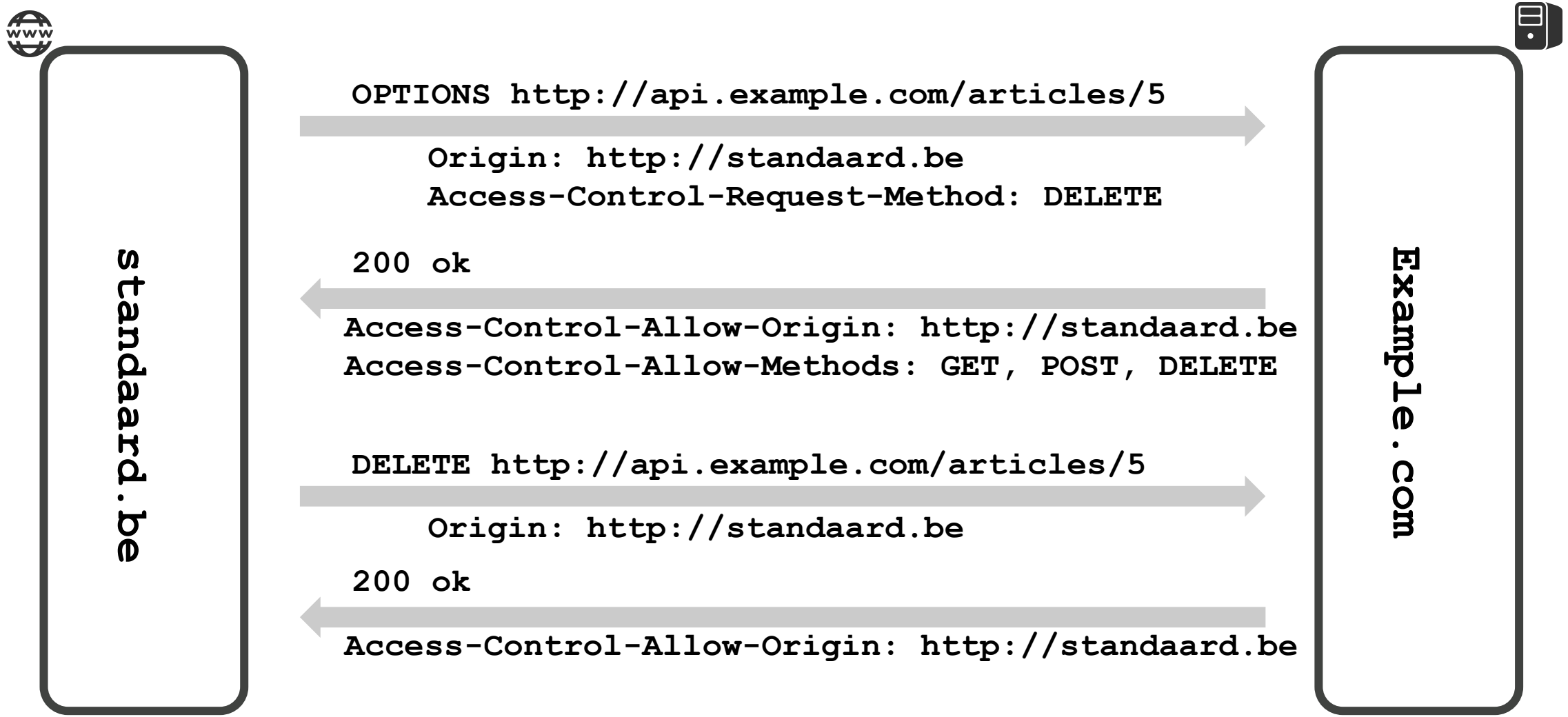
Example.com



Legacy server doesn't check for the origin header, and simply deletes the article!

CORS does not allow this

Brief Overview of CORS



CORS-enabled REST APIs

- Server: add appropriate response headers

```
var cors = require('express-cors');
app.use(
  cors( { allowedOrigins: ['http://127.0.0.1:3000'] } ));
```

- Client: Do we really have to deal with 6 new response headers?
  - No!
  - Simply consume the API like a same-origin API
    - If the server grants your app access, the browser will take care of it

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Outline

- The architecture of a single page application
  - Moving stuff from the server to the client
- Authentication and authorization
  - In combination with a stateless API
- Injection vulnerabilities and countermeasures
  - Getting rid of XSS, if you do it right
- Remote API access
  - Unintentional through CSRF, and intentional using CORS

Conclusion

- Single page applications are the next big thing
  - Great user experience, clear separation of concerns
- Their architecture empowers the client-side
  - But with great power there must also come great responsibility
- Security features are available in software
  - Choose wisely, and deploy vigorously

Securing Single Page Applications

Philippe De Ryck

philippe.deryck@cs.kuleuven.be

DistriNet

iMinds KU LEUVEN

SecureApplication Development

secappdev.org