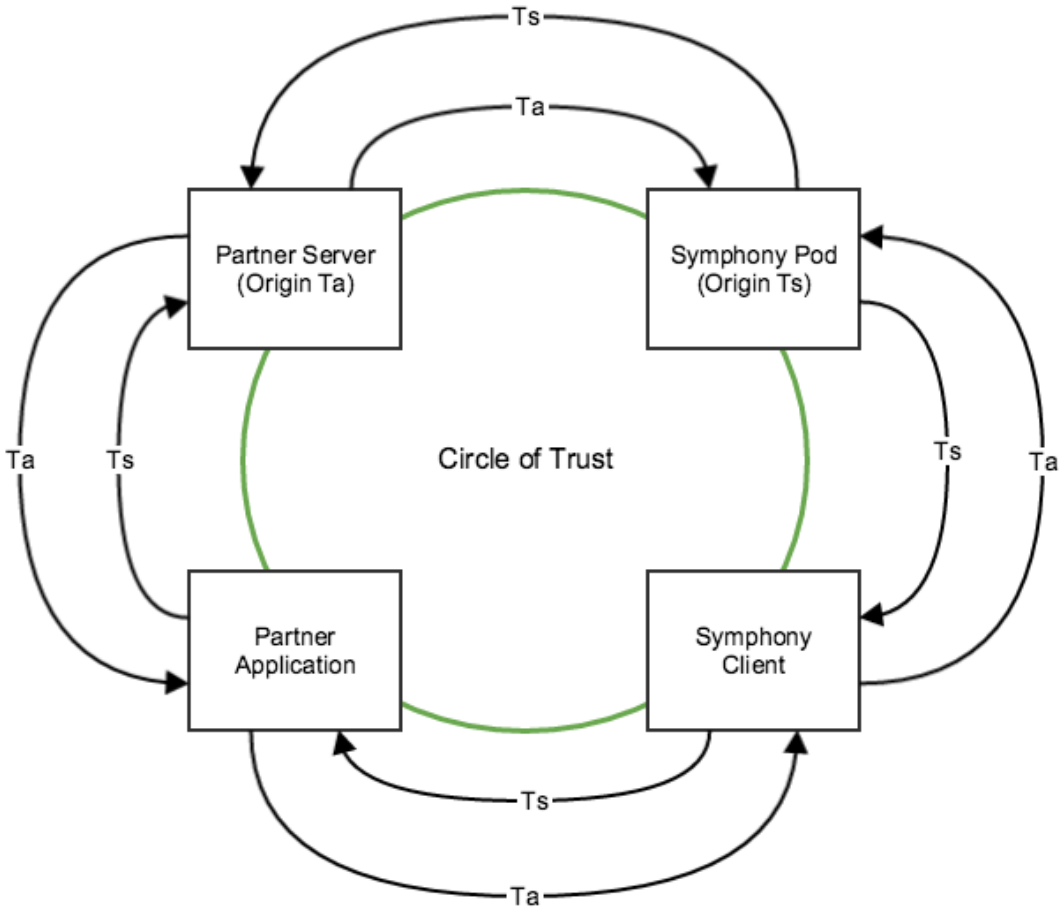
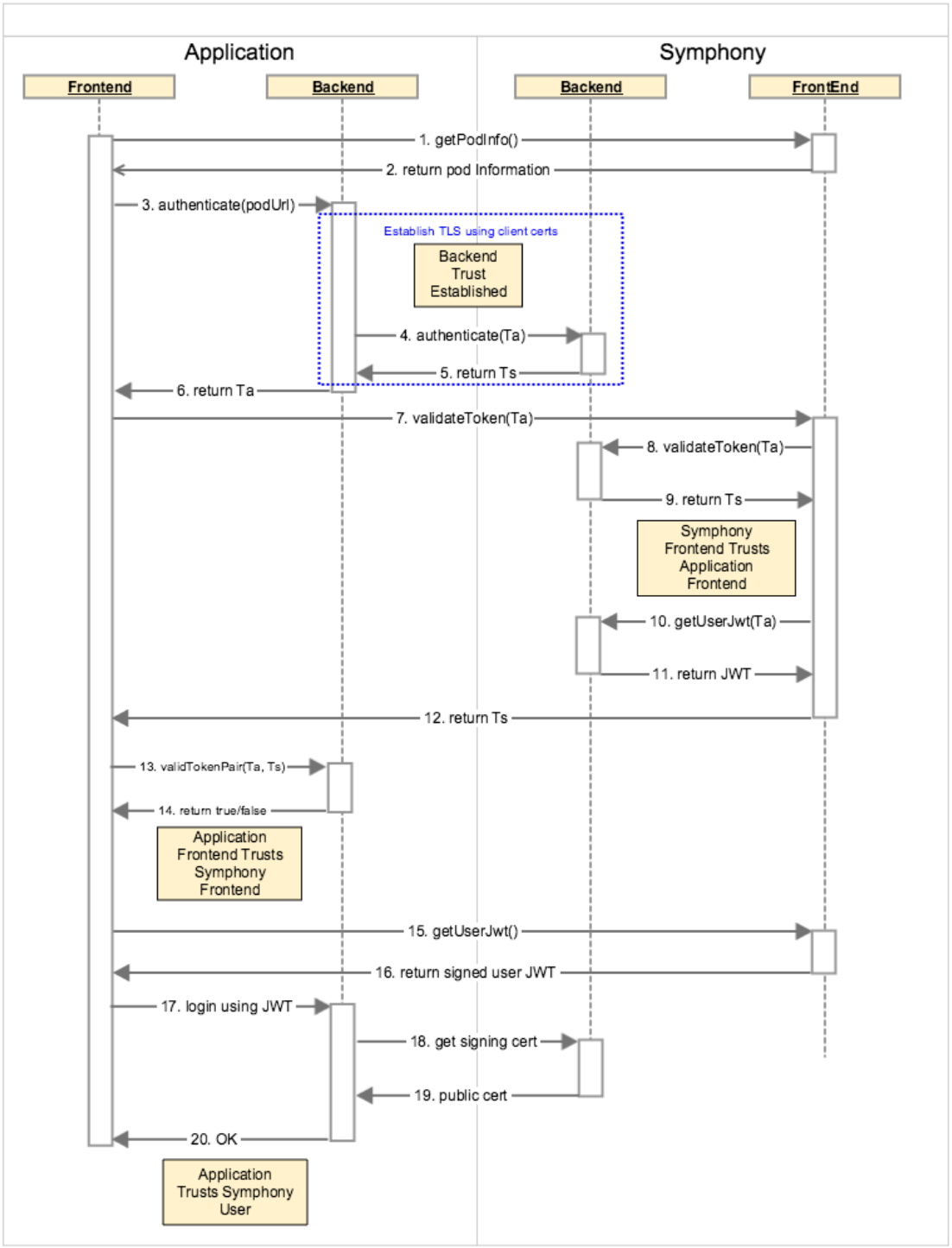
Application Authentication

The goal of application authentication is to build two-way trust between a partner's extension application and the Symphony web client. This presents a unique problem because of our security model. Trust has to be established not only between the Symphony client and the application, but the application will need to trust the Symphony client as well. This will take more than a single token such as would be provided using OAuth, since that can only be used to build trust in one direction. The authentication flow, therefore, uses two tokens T_a and T_s. These tokens are generated by the partner's backend server and the Symphony backend after those servers mutually authenticate each other using X.509 certificates which have been installed previously. A "circle of trust" is established by passing the tokens in "opposite directions" so that each backend server receives matching tokens via two separate paths.



The diagram below shows the interaction between the four "actors" in the Application Authentication flow. Note that this flow starts after the user has logged into the Symphony web client (establishing identity of the user and trust between the Symphony web client and the Symphony backend). The flow begins after the user loads the partner application.



Step	Action	Notes
1	Partner application uses Client Extension API to get info about pod	No parameters required. This action is initiated when the partner application is loaded for the first time

2	Pod information returned and includes pod application authentication URL	
3	Partner application sends request to its backend to initiate server-to-pod authentication	Pod URL that was returned in step 2 is included in request. Note that since this is a request from the partner application to the partner server and the implementation details are totally up to the partner.
4	Partner server calls the application authentication URL on the Symphony pod.	In order to call the application authentication endpoint, a TLS session using client authentication must be established. An "application user" must have been previously created on the pod (using Admin Portal) and a client certificate must have been uploaded for that user. If TLS session can be successfully negotiated, trust between partner server and the Symphony pod is established. The partner server generates and passes a token, T_a, to the Symphony backed. Symphony generates a matching token, T_s, and stores the token pair. The tokens are simple strings. The token received from the partner application is opaque to Symphony but may have meaning to the partner server. The partner server should assume the same thing for the Symphony token.
5	Pod returns matching token, T_s.	Partner server should store the token pair. They will be needed later.
6	Partner server returns T_a back to partner application.	T_a is beginning the counterclockwise journey around the circle of trust (see picture above).
7	Partner application uses Client Extension API to validate T_a	T_a continues its journey
8	Symphony request that the Pod validates T_a	T_a has now completed the circle back to the service that created it (or at least knows it). If the token is part of a current (tokens are short-lived) token pair, the matching T_s token is returned to the Symphony client.
9	Pod returns the T_s token to the Symphony Client	This verifies for the Symphony Client that the partner application is valid (talking to a trusted partner server). The T_s token is now beginning the clockwise journey back to the partner's backend server.
10	Symphony client requests user JWT from pod	The JWT contains information identifying the current user and is signed using a Symphony certificate whose public key is accessible via the internet. The exact contents of the JWT may vary between applications; some applications only receive a user reference ID while others receive enough information to uniquely identify the user.
11	JWT returned to Symphony Client	The Symphony Client holds on to the JWT in case the parter application wants it later to positively identify the user.
12	The Symphony Client returns the T_s token to the partner application	

13	Partner client passes the T_s (and maybe the T_a) token back to partner server.	This request is between the partner application and the partner server. It's implementation details are up to the partner. For instance, if the partner application and server maintain a sticky session, only the T_s token would need to be sent back since the server would already have the corresponding T_a token saved in the session. If the token pair matches a known set of tokens, then the clockwise route of T_s is complete and partner backend knows that its client application is talking to a valid Symphony Client.
14	Partner server tells partner application that the token is valid	This confirms for the partner client that is it talking to a valid Symphony Client.
15	Partner client uses Client Extension API to get user JWT	The Symphony Client will not return the JWT if trust has not been established by executing at least steps 1 - 11.
16	User JWT returned	The JWT is signed by the Symphony backend to ensure that it is not tampered with. The signature can be verified using publicly available certificate on the Symphony pod. The JWT contains information that identifies the user (e.g. userReferenceId, user ID, email address, first/last/display name, company, username, etc.)
17	Partner client authenticates the user to the partner backend.	Note that since this is a request from the partner application to the partner server and the implementation details are totally up to the partner.
18	Partner server gets public signing certificate from Symphony pod	
19	Public signing cert returned	At this point the partner server can both verify that the JWT has not been tampered with and decode the JWT to get the user's identity.
20	Partner server acknowledges successful authentication	At this point the partner application trusts that the user is who they say they are

JavaScript Client Authentication API

The existing startup sequence as currently documented will be modified to incorporate this two way authentication.

`SYMPHONY.application.hello()`

This method has been modified to return the pod id of the current pod. It returns a promise that will be fulfilled with an object with these members:

- `theme` deprecated
- `themeV2` is the theme information for the Web Client
- `pod` is the current pod id. This can be used to lookup the correct back-end end point.

`SYMPHONY.application.register(auth, servicesWanted, servicesSent)`

Call this method to register the application with the Symphony web client. This method will call the backend to validate the passed application id and token. The backend will return the application token.

`auth` is either the id of the application for insecure apps, or an object with these members:

- `appId`: the unique identifier of the application

- `tokenA`: the application token generated as part of the authentication steps

This method returns a promise that will be fulfilled with an object with these members:

- appId: the provided application identifier
- tokenS: the Symphony token generated as part of the authentication steps

If authentication fails, this method will reject the promise.

`getJwt()`

A new service is made available to get a JWT with information about the current user. The name of the service is '`extended-user-info`'. Call the method '`getJwt`' to get a promise that will be resolved with the JWT containing user information. This JWT has been signed with the pod's private key, and can be verified using the pod's public key.

JWT Payload Format

```
{
    "aud" : "<id of app>",
    "iss" : "Symphony Communication Services LLC.",
    "sub" : "<Symphony user ID>",
    "exp" : "<expiration date in millis>",
    "user" : {
        "id" : "<Symphony user ID>",
        "emailAddress" : "<email address>",
        "username" : "<Symphony username>",
        "firstName" : "<first name>",
        "lastName" : "<last name>",
        "displayName" : "<display name>",
        "title" : "<title>",
        "company" : "<company>",
        "companyId" : "<company (pod) ID>",
        "location" : "<location>",
        "avatarUrl" : "<URL for user's avatar>",
        "avatarSmallUrl" : "<URL for user's small avatar>"
    }
}
```

Backend API

Backend authentication

For the backend-to-backend application authentication, the application backend can invoke this REST-full API to the Symphony Backend using the application certificate. The T_a is generated by the application and must be **unique** for each new application authentication request. The Symphony backend will generated a unique corresponding T_s for this T_a pair. This T_s will live in the Symphony Backend for a TTL of ~ 5 mins for subsequence validateToken call in Step 7 of the Authentication Flow diagram above. If the T_s has expired, the application will need to re-authenticate with the Symphony backend , starting with Step 3 of the Authentication Flow diagram above.

```
POST https://<host>:<port>/sessionauth/v1/authenticate/extensionApp
{
  "appToken": <Ta>   // String
}


Response:
200 OK
{
 "appId": <appId>,       // String
 "appToken": <Ta>,       // String
 "symphonyToken": <Ts> // String
}
```

Pod Certificate

To validate the authenticity of the JWT token return in Step 16 of the Authentication flow diagram, the application can call this API to retrieve the public certificate that the Symphony Backend used to sign the JWT token. This API does not required an application certificate to retrieve the pod public certificate.

```
GET https://<host>:<port>/sessionauth/v1/app/pod/certificate


Response:
200 OK
{
 "certificate": <public certificate in PEM format> // String
}
```